On Querying Versions of Multiversion Data Warehouse [*]

Tadeusz Morzy
Poznan University of Technology
Institute of Computing Science
Poland
tmorzy@cs.put.poznan.pl

Robert Wrembel
Poznan University of Technology
Institute of Computing Science
Poland
rwrembel@cs.put.poznan.pl

ABSTRACT

A data warehouse (DW) is fed with data that come from external data sources that are production systems. External data sources, which are usually autonomous, often change not only their content but also their structure. The evolution of external data sources has to be reflected in a DW, that uses the sources. Traditional DW systems offer a limited support for handling dynamics in their structure and content. A promising approach to handling changes in DW structure and content is based on a multiversion data warehouse. In such a DW, each DW version describes a schema and data at certain period of time or a given business scenario, created for simulation purposes. In order to appropriately analyze multiversion data, an extension to a traditional SQL language is required. In this paper we propose an approach to querying a multiversion DW. To this end, we extended a SQL language and built a multiversion query language interface with functionality that allows: (1) expressing queries that address several DW versions and (2) presenting their results annotated with metadata information.

Categories and Subject Descriptors

H.2 [**Database Management**]: Languages - query languages

General Terms

Design, Languages, Experimentation

Keywords

data warehouse, data versioning, schema versioning, multiversion query, metadata

[*]This work is partially supported by the grant no. 4 T11C 019 23 from the Polish State Committee for Scientific Research (KBN), Poland

1. INTRODUCTION

A data warehouse (DW) integrates autonomous and heterogeneous external data sources (EDSs) and makes the integrated information available for analytical processing, decision making, and data mining applications. As external data sources are autonomous, they may evolve in time independently of each other and independently of a DW that integrates them [22]. Changes in EDSs can be categorized as: (1) content changes, i.e. insert/update/delete data, and (2) schema changes, i.e. add/modify/drop an attribute or a table. A way to tackle the problem of content and schema changes in EDSs is to ensure correct propagation of these changes to a DW. A DW schema adjustments can be done in two different ways, namely schema evolution and schema versioning. The first approach consists in updating a schema and transforming data from an old schema into a new schema. Only the current version of a schema is present. In contrast, the second approach keeps track of the history of all versions of a schema. Versioning can be done implicit by temporal extension or explicit by physically storing different versions of data.

The process of good decision making often requires forecasting future business behavior, based on present and historical data as well as on assumptions made by decision makers. This kind of data processing is called the *what-if analysis.* In this analysis, a decision maker simulates in a DW changes that might happen in real world, creates virtual possible scenarios, and explores them with OLAP queries. To this end, a DW must provide means of creating various DW alternatives, represented by different DW versions.

A DW capable of managing its multiple versions will further be called a *multiversion data warehouse* (MVDW). In order to analyze data stored in a MVDW, new analytical tools and an extended query language are required. Such a query language has to be capable of extracting partial results from versions of interest, integrate them into one consistent result set (if possible), as well as present partial and integrated results in a meaningful form to a user.

1.1 Our approach and contribution

In our approach, changes in EDSs are handled in versions of a data warehouse. A DW administrator creates a version of a DW explicitly and this version represents the structure and content of EDSs within a given period of time. Maintaining versions of the whole DW allows us:

- to manage explicitly different DW schemas and associated data, pertinent to given periods in past;

- to run queries that span multiple versions and compare their results often without the need of excessive transformations of data, unlike in the case of temporal approaches (e.g. [8, 9]);
- to create and manage various alternative virtual business scenarios required for the what-if analysis.

In order to query a multiversion DW and present query result to a user, we propose an extension to a traditional SQL language with functionality of traversing several versions, comparing their results, and then merging them, if needed and possible, to a common set of data. In our approach and prototype system, a query on a MVDW that addresses more than one version is processed in two following steps. In the *first step* a query is decomposed into the set of independent *partial queries*, each for one DW version specified in an original query. Every partial query is executed in its own DW version. Next, the result of every partial query is presented to a user as a separate result, annotated with version and metadata information. This information allows to analyze and interpret the obtained results appropriately. In the *second step*, partial results are integrated into a common set of data, if possible. In order to support our querying technique we have built:

- a graphical user interface for visualising MVDW objects and composing multiversion queries;
- a SQL parser for parsing multiversion queries;
- a SQL executor for executing partial queries in appropriate DW versions, receiving data and combining them into one consistent result set;
- a query result visualizer for presenting partial and integrated query results.

Our prototype software was implemented in Java and Oracle PL/SQL language. Data and metadata are stored in an Oracle9i database.

1.2 Basic Definitions

A DW takes advantage of a multidimensional data model [10, 12, 13, 18] with **facts** representing elementary information being the subject of analysis. A fact contains features, called **measures**, that quantify the fact and allow to compare different facts. Values of measures depend on a context set up by **dimensions**. Typical examples of dimensions include *Time*, *Location*, *Product*, etc (cf. Figure 1). Dimensions are usually organized in **hierarchies**, e.g. *Product* in Figure 1. A schema object in a dimension hierarchy is called a **level**, e.g. *Product* and *Vat_Category*. A dimension hierarchy specifies a way measures are aggregated. A lower level rolls-up to an upper level, yielding more aggregated data. Values in every level are called **level instances**. Level instances have a structure set up by level schemas. A **level schema** defines the name of the level and the set of its attributes with their domains. Similarly, values of a fact are called **fact instances**. Fact instances have a structure set up by a fact schema. A **fact schema** defines the name of the fact and the set of its attributes and their domains.

Every dimension has its structure defined by a **dimension schema** that includes: (1) the name of the dimension, (2) the set of levels in the dimension, and (3) hierarchical assignments between levels. For example, dimension *Product* has the following schema $\{Product,\ Product \rightarrow Vat\_Category\}$, where $\rightarrow$ represents hierarchical assignment between levels, i.e. *Product* rolls-up to *Vat_Category*.

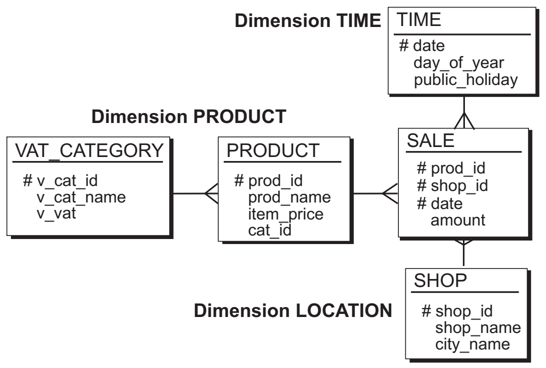


Figure 1: An example DW schema on sale of products

A **dimension instance** of dimension D is composed of hierarchically assigned instances of levels in D, where the hierarchy of level instances is set up by the hierarchy of levels. For example the *Product* dimension instance includes $\{Ytong\ brick \mapsto 7\%\ vat\}$, where $\mapsto$ represents hierarchical assignment between level instances. The values of dimension instances and hierarchies they form constitute a **dimension instance structure**.

The structure of information being stored in a DW is described by a **DW schema**, composed of dimension schemas and fact schemas. Whereas a **DW instance** consists of dimension instances and fact instances.

Paper organization. The rest of this paper is organized as follows. Section 2 discusses existing approaches to handling changes in the structure of a DW. Section 3 overviews our concept of a multiversion DW. Section 4 presents SQL extensions and a prototype interface for querying a MVDW. Section 5 outlines problems related to querying heterogeneous DW schema versions. Section 6 presents the metamodel of the MVDW. Section 7 summarizes the paper.

2. RELATED WORK

The approaches to the management of changes in a DW can be classified into the two following categories that support: (1) schema and data evolution: [5, 11, 12, 23, 16, 14], (2) temporal and versioning extensions [7, 8, 9, 19, 15, 17, 21, 22, 2, 6, 18, 1].

The approaches in the first category support only one DW schema and its instance. When a change is applied to a schema all data described by the schema must be converted to a new structure, that incurs high maintenance costs. In these approaches data coming from different periods of time are timestamped and stored together in the same data structure. This feature limits the set of schema changes that can be handled.

In the approaches from the second category, in [7, 8, 9, 19] changes to a DW schema are time stamped in order to create temporal versions. However, [7] and [19] expose their inability to express and process queries that span or compare several temporal versions of data. On the contrary, the model and prototype of a temporal DW presented in

[8, 9] support queries for a particular temporal version of a DW or queries that span several versions. In the latter case, conversion functions must be applied, as data in temporal versions are virtual.

In [15, 17, 21, 22] implicit versions of data are used for avoiding conflicts and mutual locking between OLAP queries and transactions refreshing a DW. As versions are implicitly created and managed by a system, these mechanisms can not be used in the what–if analysis. The same drawback applies to the previously discussed temporal DW that can manage only consecutive versions that are linearly ordered by time.

On the contrary, [2] proposes permanent user defined versions of views in order to simulate changes in a DW schema. However, the approach supports only simple changes in source tables and it does not deal either with typical multidimensional schemas or evolution of facts or dimensions. Also [6] supports permanent time stamped versions of data. The proposed mechanism, however, uses one central fact table for storing all versions of data. In a consequence, the set of schema changes that may be applied to a DW is limited, and only changes to dimension schema and dimension instance structure are supported. The paper by [18] analyzes updates to dimensions and proposes consistency criteria that every dimension has to fulfill. It gives an overview how the criteria can be applied to a temporal DW only.

An approach supporting the what-if analysis was presented in [1]. It may be considered as a kind of virtual versioning. A hypothetical query is executed on a virtual structure, called scenario. Then, the system using substitution and query rewriting techniques transforms the hypothetical query into an equivalent query that is run on a real DW. As this technique computes new values of data for every hypothetical query, based on virtual modifications, a user will experience performance problems in large DWs.

Commercial DW systems and OLAP tools existing on the market (e.g. Oracle9i/10g, Oracle Express Server, IBM DB2, Sybase Adaptative Server Enterprise, Ingres DecisionBase OLAP Server, NCR Teradata, Hyperion Essbase OLAP Server, SAP Business Warehouse) support neither managing changes of a DW structure, nor the what-if analysis functionality, nor querying multiversion data. The exception is SAP Business Warehouse, that is capable of handling only simple changes in dimension data. Oracle's what-if analysis allows to create only the simplest hypothetical rankings of records or expressing hypothetical analysis just in a query using the `model` clause [24].

3. MULTIVERSION DATA WAREHOUSE

This section informally overviews our concept of a multiversion DW. Its formal description was presented in [20]. In our approach, changes to a schema may be applied to a new version of a DW. This version, called a child version, is derived from a previous version, called a parent version. Versions of a DW form a version derivation graph. Each node of this graph represents one version, whereas edges represent **derived–from** relationships between two consecutive versions. A version derivation graph is a DAG. A **multiversion data warehouse** is composed of the set of its versions. Every version DWV_i of a MVDW is in turn composed of a schema version S_i and an instance version I_i that stores data described by S_i.

3.1 Types of Versions

We distinguish two following kinds of DW versions: real versions and alternative versions. A **real version** reflects changes in the real world. Real versions are created in order to keep up with the changes in a real business environment, like for example: changing organizational structure of a company, changing geographical borders of regions, opening and closing shops, changing prices/taxes of products. Real versions are linearly ordered by the time they are valid within. The main purpose of maintaining **alternative versions** is to support the what-if analysis, i.e. they are used for simulation purposes. An alternative version is created from a real version or from an alternative one. Several alternative versions may be created from the same parent version.

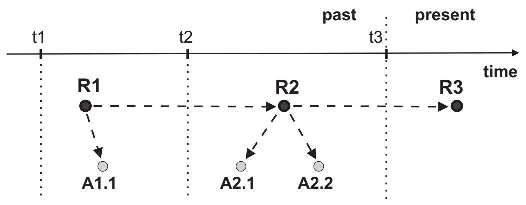


Figure 2: Real and alternative data warehouse versions ordered by time and derivation relationships

Figure 2 schematically shows real versions and alternative versions. *R1* is an initial real version of a DW. Based on *R1*, a new real version *R2* was created. Similarly, *R3* was derived from *R2*. *A1.1* is an alternative DW version derived from *R1*, whereas *A2.1* and *A2.2* are alternative versions derived from *R2*. Note that real versions are linearly ordered, whereas alternative versions may branch.

Every version is valid within certain period of time. In order to check a version validity, every real and alternative DW version has associated, so called *validity time*, represented by two timestamps, i.e. *begin validity time* (BVT) and *end validity time* (EVT) (cf. [3] for details). For example, real version *R1* (from Figure 2) is valid within time t1 (BVT) and t2 (EVT), *R2* is valid within t2 and t3, whereas *R3* is valid within t3 until present. Alternative versions *A2.1* and *A2.2* are valid within the same time period as their parent real version *R2*.

3.2 Changes to a DW schema

We distinguish two following groups of elementary operations that modify a data warehouse schema: (1) operations that change the schema of a DW (further called schema change operations) and (2) operations that change the structure of dimensions (further called dimension structure change operations). All the above operations address one particular version of a data warehouse. We will outline the operations in the context of a ROLAP server, which is our implementation environment.

Schema change operations include: (1) creating a new level table with a given structure, (2) connecting a given level table with its sub- and superlevel tables, (3) disconnecting a given level table from its dimension hierarchy, (4) removing a previously disconnected level from a schema, (5) adding a new attribute to a level, (6) dropping an attribute from a level, (7) changing a domain of a level attribute, (8) creating a new fact table, (9) adding a new attribute into a fact table, (10) associating a given fact table with a given dimension, (11) removing a non primary key or non foreign key attribute from a given fact table, (12) removing an association (a foreign key) between a fact table and a dimension, (13) removing a fact table, previously disconnected from a schema, (14) renaming an attribute, (15) renaming a table.

Operations: 2, 3, 9, 10, 11, 12, 13, 14, 15 cause that: either (1) user analytical queries need to be modified in order to be applicable to a DW schema after change or (2) previous data are lost or have to be transformed to a new structure. Therefore, our prototype system suggests applying the operations to a new DW version and, if accepted by a DW administrator, a new DW version is automatically created. Other operations can be applied either to an existing DW version or to a newly created one, depending on an administrator's decision.

Dimension instance structure change operators include: (1) inserting a new level instance into a given level, (2) deleting an instance of either a top, or an intermediate, or a bottom level, (3) changing the association between a sublevel instance and its superlevel instance, (4) merging several instances of a given level into one instance of the same level, (5) splitting a given instance of a given level into several instances of the same level.

Since the operations have an impact on results obtained form analytical queries, c.f. [4] they are applied to a new DW version.

Based on the concept described above, we have implemented a MVDW that manages versions of its schemas and instances, as shortly presented in [3]. An administrator uses a graphical interface implemented in Java for deriving and modifying DW versions.

4. QUERYING MULTIVERSION DATA

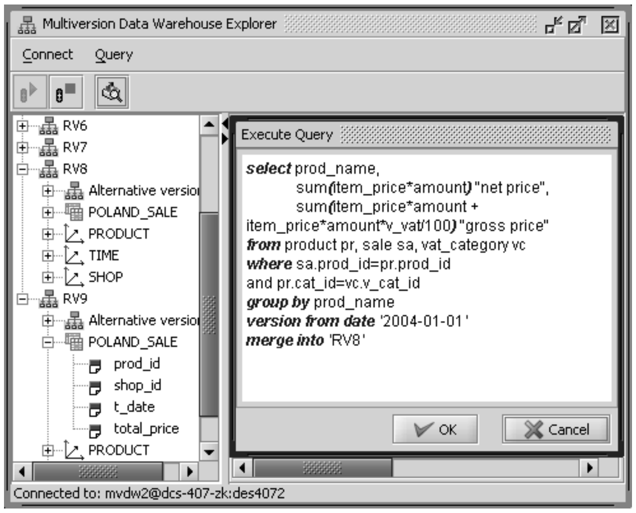
In a MVDW data of interest are usually distributed among several versions. A user may not be aware of data location and he/she can query the whole multiversion DW or a particular version or the set of versions, either real or an alternative ones. In our approach and prototype system, at the system level, querying a MVDW is done in two steps. In the **first step**:

- a multiversion query expressed by a user is decomposed to the set of independent *partial queries*, each one for a separate DW version specified in an original query;
- every partial query is then executed in its appropriate DW version;
- the result of every partial query is then presented to a user as a separate partial result annotated with version information and metadata information. The metadata information attached to partial query results allows to analyze and interpret the obtained results appropriately (c.f. Section 5).

In the **second step**, the module responsible for executing multiversion queries can combine the independent results, if possible, into a common set of data (one cube, for example).

4.1 SQL Extensions

While querying a MVDW a user has three following possibilities: either to query the current version, or to query the set of real DW versions, or to query the set of selected alternative DW versions. A query that addresses a single DW



```
select prod_name,
        sum(item_price*amount) "net price",
        sum(item_price*amount +
item_price*amount*v_vat/100) "gross price"
from product pr, sale sa, vat_category vc
where sa.prod_id=pr.prod_id
and pr.cat_id=vc.v_cat_id
group by prod_name
version from date '2004-01-01 '
merge into 'RV8'
```

Figure 3: User interface for querying a MVDW

version will further be called a *monoversion query*, whereas a query that addresses the set of versions, either real or alternative, will be called a *multiversion query*. In order to query a MVDW and present query results to a user, we propose an extension to traditional SQL language with functionality of traversing several versions, comparing their results, and then merging them, if needed and possible, into a common result set.

4.1.1 *Querying current DW version*

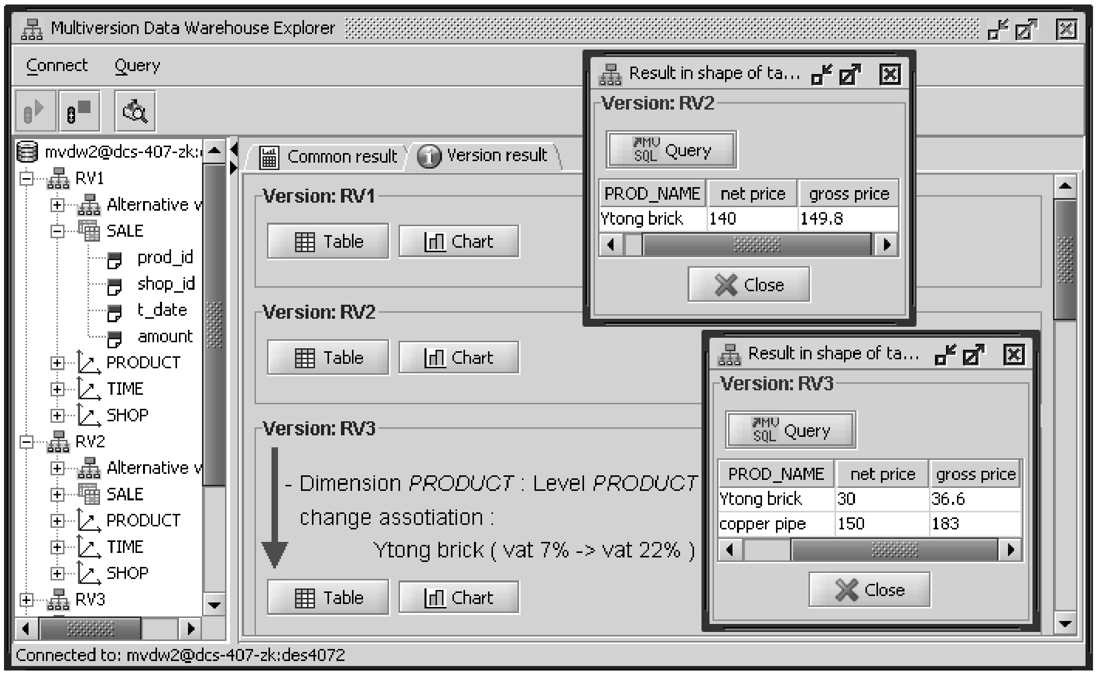
By default, a user issues monoversion queries addressing the current (latest) real DW version. In this case, from a user point of view, no extension to a SQL query language is required. However, from the implementation point of view, a query parser must be capable of finding all data belonging to the current version, since in the prototype implementation common data are shared by several DW versions (cf. [3]).

4.1.2 *Querying the set of real DW versions*

Multiversion queries on the set of *real* DW versions are defined by specifying time period of interest, real versions are valid within. To this end, the `select` command was extended with the `version from date 'begin date' to date 'end date'` clause. As pointed in Section 3.1, every version has its *begin validity time* as well as *end validity time*. These two times are used for selecting appropriate versions of interest, according to the dates specified in the `version from .. to` clause. A query is decomposed by our query parser into the set of partial queries, each for one DW version that is valid within in the specified period of time.

4.1.3 *Querying the set of alternative DW versions*

Since alternative versions are not linearly ordered by time and may branch, a user has to explicitly provide a set of alternative versions of interest. To this end, the `select` command was extended with the `alternative version in ({ver_id | ver_name},..., )` clause, where `ver_id` and `ver_name` represent a DW version identifier and name, respectively.



- Dimension *PRODUCT* : Level *PRODUCT*
change assotiation :
Ytong brick (vat 7% -> vat 22%)

Version: RV2

PROD_NAME	net price	gross price
Ytong brick	140	149.8

Version: RV3

PROD_NAME	net price	gross price
Ytong brick	30	36.6
copper pipe	150	183

Connected to: mvdw2@dcs-407-zk:des4072

Figure 4: Query results visualizer - an example multiversion query result annotated with metadata information

4.1.4 *Querying a single DW version*

Monoversion queries on either a real or an alternative DW version, are treated as special cases of queries discussed in Sections 4.1.2 and 4.1.3 and are defined by specifying in the `version` clause either dates that a real version is validi within or an alternative version identifier/name.

4.1.5 *Merging results of partial queries*

By default, every result set of a partial query is presented to a user separately. In some cases, however, partial queries can be merged into one result set (one cube), cf. Section 5.3. Merging the results obtained by partial queries is defined by including the `merge into {ver_id | ver_name}` clause, where `ver_id` or `ver_name` point to a DW version whose schema will be used as a destination schema for all the obtained partial query results. Since original partial results have to be transformed into a common schema, transformation methods have to exist in the MVDW data dictionary. These methods, defined by a DW administrator, are responsible for transforming data between adjacent DW versions.

4.2 Query interface and results visualizer

Our prototype system has a query interface that allows to construct queries for a multiversion DW and present their results. The prototype is written in Java and Oracle PL/SQL languages, whereas data and metadata are stored in an Oracle9i database. Its main management window is shown in Figure 3. The left hand side panel - an *object navigator* allows browsing through versions of a MVDW. The schema of every DW version can be explored there. The left hand side panel is used for constructing queries as well as for presenting their results. In the current beta version of the prototype, a multiversion query is constructed by a user by writing it directly in an interpreter window, as shown in Figure 3. In the next release we are going to replace it with a graphical multiversion query builder. The results of multiversion queries are presented in the *results visualizer* as: (1) partial result sets of every partial query, and (2) an integrated result, if an integration is possible (cf. Section 5.3). Additionally every partial result set is annotated with version informatin and metadata information about schema and instance structure changes in a DW version being addressed by a query. An example output of the query results visualizer is shown in Figure 4. Every result set is displayed in its own window. In the example window, two result sets are presented, one from version *RV2* and one from *RV3.* Additionally result set from *RV3* is annotated with metadata information describing a change of the *Product* dimension instance structure. The *Common results* tab allows browsing through an integrated result set.

4.3 Prototype limitations

The current beta version of the prototype multiversion query interface has the following limitations:

- all predicates of the `select` command apply to all DW versions pointed to in the `version from` and `version in` clauses, i.e. it is not possible to express a predicate on a single DW version;
- the query parser is unable to infer appropriate versions of interest from the `where` clause;
- a query parser is able to compute an integrated result set of a multiversion query using simple aggregate functions (`sum`, `min`, `max`, `avg`), but not using the advanced OLAP functions, like for example window functions (cf. [24]).

5. HETEROGENEOUS SCHEMA VERSIONS - CASE STUDIES

Merging results of partial queries into a common DW version will be possible if a multiversion query addresses attributes that are present in all versions of interest and if there exist transformation methods between adjacent DW versions (if needed). For example, it will not be possible to merge the result sets of a multiversion query on DW version 1 and 2, computing the sum of products sold, if in version 1 attribute *amount* (cf. Figure 1) exists and in version 2 the attribute was dropped.

Two or more tables, either fact or dimension level, may have identical schemas (i.e. identical table names, identical number of attributes, their names, and domains, identical integrity constraints), in some DW versions. Such tables will be called *homogeneous* in these versions. Tables that differ with respect to their schemas in some DW versions will be called *heterogeneous* in these versions. Furthermore, two dimension instances di_{V1} and dj_{V2} will be called *homogeneous* in DW versions *V1* and *V2* if: (1) their dimension level tables are homogeneous in *V1* and *V2* and (2) they the have identical their dimension instance structure (c.f. Section 1.2) in these DW versions. For example, the following instance of the *Product* dimension $\{$*Ytong brick*$\mapsto$*7% vat*$\}$ in DW version *V1* and the instance $\{$*Ytong brick*$\mapsto$*22% vat*$\}$ in DW version *V2* are heterogeneous since their dimension instance structure (classification to vat categories) is different in these versions. Dimensions that differ with respect to their instance structure and/or use heterogeneous dimension level tables in some DW versions will be called *heterogeneous* in these versions.

The fact that tables and whole DW schemas are homogeneous or not, has an impact on query parsing as well as on the possibility of merging the results of partial queries. While parsing and executing multiversion queries and and merging partial result sets, the query interpreter and executor have to consider several cases including heterogeneous and homogeneous schemas. In the paper we outline here two cases when: (1) fact tables are homogeneous and dimension instance structure is heterogeneous and (2) fact tables are heterogeneous and dimension instance structure is homogeneous.

5.1 Homogeneous fact tables and heterogeneous dimension instance structures

In order to illustrate changes to dimensions resulting in heterogeneous dimension instance structures, let us consider a DW schema from Figure 1 and let us assume that initially, in February 2003 in real version *RV2*, there existed 3 shops, namely *Castorama*, *Praktiker*, and *Marx Pipes*. These shops were selling *Ytong bricks* with 7% of VAT.

Case 1: Reclassifying level instances. Let us assume that in March, *Ytong bricks* were reclassified to 22% VAT categoriy (which is a real case of Poland after joining the European Union). This reclassification was reflected in a new real DW version *RV3*.

The discussed changes are shown in Table 1 presenting the content of the *Product* table in version *RV2* and *RV3*. The *sys_id* attribute is a system record identified used at the implementation level and not visible to a user. It is used for managing changes in dimension instances structure, e.g. reclassification, splitting, and merging.

Table 1: An example content of the *Product* table in version *RV2* and *RV3*

Product RV2			
sys_id	prod_id	prod_name	prod_cat_id
700	1	Ytong bricks	vat_7

Product RV3			
sys_id	prod_id	prod_name	prod_cat_id
800	1	Ytong bricks	vat_22

Every change to a DW schema is registered in the multiversion data warehouse data dictionary (cf. Section 6 and Figure 5) whose entries are used for multiversion query processing as well as for annotating query results. An information about all levels is stored in the *Levels* data dictionary table. Attribute *lev_map_tab_name* stores the name of a table whose content provides mappings between a dimension level instance in a parent DW version V_i and related to it dimension level instance in a child DW version V_j, directly derived from V_i. The mapping table is generated and maintained automatically by the prototype system for every level table whose instance changed between two adjacent versions. The structure of this table is as follows: `{old_tab_name, old_sys_id, new_tab_name, new_sys_id}`. Attributes `old_tab_name` and `new_tab_name` store the names of a level table in a parent and a child DW version, respectively. These attributes allow to handle changes in the names of level tables between DW versions. Attributes `old_sys_id` and `new_sys_id` store system identifiers of level instances being mapped form a parent to a child DW version, respectively.

In our example, level *Product* in version *RV3* will have associtated *Product_RV3_RV2_Map_Table* with the content shown in Table 2, which means that: a product identified by 700 in version *RV2* is represented by product 800 in version *RV3*, cf. Table 1. The name of this *_Map_Table* will be stored in a record describing level *Product* in DW version *RV3*, inserted into the *Levels* dictionary table.

Table 2: An example content of the *Product* level mapping table (*Product_RV3_RV2_Map_Table*)

old_tab_name	old_sys_id	new_tab_name	new_sys_id
Product	700	Product	800

Now we may consider a user query that addresses DW versions from January till March, i.e. *RV1*, *RV2*, and *RV3* and computes net and gross total sale of products. The query is decomposed into three queries: one query for one DW version. Then the partial results are returned to a user with metadata information describing changes in dimension instance structures. In the current implementation of our prototype, the metadata information is presented in a text form. For example, the partial result of a query addressing version *RV3* will be annotated with:
Dimension PRODUCT: Level PRODUCT:
change association:
Ytong bricks(vat 7% $\rightarrow$ *vat 22%)*
as shown in Figure 4. This way a sale analyst will know that

a gross sale increase form February to March was at least partially caused by VAT increase.

Case 2: Merging or splitting level instances. In order to illustrate this case, let us assume that in June, shops *Castorama* and *Marx Pipes* were merged into one shop *Castorama* in a new real DW version *RV6*. The discussed changes are shown in Table 3 presenting the content of the *Shop* table in version *RV5* and *RV6*.

Table 3: An example content of the *Shop* table in versions *RV5* and *RV6*

Shop RV5			
sys_id	shop_id	shop_name	city_name
100	1	Castorama	Poznan
200	2	Praktiker	Warsaw
300	3	Marx Pipes	Poznan

Shop RV6			
sys_id	shop_id	shop_name	city_name
400	1	Castorama	Poznan
200	2	Praktiker	Warsaw

In our example, level *Shop* in version *RV6* will have associated the *Shop_RV6_RV5_Map_Table* with the content shown in Table 4, which means that: (1) a shop identified by 200 has not changed between versions *RV5* and *RV6*, (2) two shops, namely those identified by 100 and 300 in version *RV5*, constitute a shop identified by 400 in version *RV6*.

While querying version *RV6* a user will obtain a partial result set annotated with the following metadata information informing about merging shops:
Merge(Castorama, Marx Pipes) → *Castorama*

Splitting level instances will be handled similarly as merging, by registering splitted instance in an appropriate *_Map_Table*.

Table 4: An example content of the *Shop* level mapping table (*Shop_RV6_RV5_Map_Table*)

old_tab_name	old_sys_id	new_tab_name	new_sys_id
Shop	200	Shop	200
Shop	100	Shop	400
Shop	300	Shop	400

Case 3: Level detachment or inclusion. In order to illustrate this operation let us consider a DW schema from Figure 1, existing in version *RV6* and having two records, as shown in Table 3. Let us further assume that in version *RV7* a new level, named *City* was created as a superlevel of *Shop* by moving attribute *Shop.city_name* out from the *Shop* table.

In order to handle this case, firstly an information about level detachment has to be registered in the data dictionary, and secondly, appropriate *Shop* level instances from version *RV6* have to be mapped to their corresponding level instances in version *RV7*.

The data dictionary has to register information about: (1) newly created level *City* and modified level *Shop* in version *RV7* - it is stored in the *Levels* data dictionary table (cf. Section 5), (2) attributes of levels *City* and *Shop* - it is stored in the *Attributes* table, (3) a mapping between the *Shop* level in version *RV6* and *RV7* - it is stored in the *Levels_Mappings* table, (4) mappings between attributes of level *Shop* in version *RV6* and levels *Shop* and *City* in version *RV7* - it is stored in the *Attr_Mappings* table. An example content of the *Levels* table representing the discussed modifications are shown in Table 5. The entries to the *Levels_Mappings* table contain two following records: (1) a record mapping *Shop* in version *RV6* to *Shop* in version *RV7*, (2) a record mapping *Shop* in version *RV6* to *City* in version *RV7*, since attribute *Shop.city_name* was moved to level *City*.

Table 5: An example content of the *Levels* data dictionary table

lev_id	lev_name	lev_map_tab_name
shop_v6	shop	
shop_v7	shop	Shop_RV7_RV6_Map_Table
city_v7	city	City_RV7_RV6_Map_Table

The entries to the *Attr_Mappings* table will be as follows. Firstly, attributes *Shop.shop_id* and *Shop.shop_name* from version *RV6* are mapped to *Shop.shop_id* and *Shop.shop_name* in version *RV7*, respectively. Secondly, attribute *Shop.city_name* in version *RV6* is mapped to *City.city_name* in version *RV7*.

The instances of modified levels have also to be mapped from version *RV6* to *RV7*. To this end, two mapping tables are used. Their names are stored as values of attribute *Levels.lev_map_tab_name*, as shown in Table 5. *Shop_RV7_RV6_Map_Table* stores mappings between *Shop* instances in version *V7* and *V6*, whereas *City_RV7_RV6_Map_Table* stores mappings between instances of *Cities*. Their contents are shown in Table 6 and have the following meaning. Shops from version *RV6* identified by 400 and 200 are represented in version *RV7* by shops identified by 500 and 600, respectively. The last two shops are newly created in version *RV7* and they do not contain the value of attribute *city_name*. Shops from version *RV6* identified by 400 and 200 are mapped in version *RV7* to instances of *City*, identified by 10 and 20 respectively. It is because values of *Shop.city_name* were moved into these instances.

Table 6: An example content of the *Shop* and *City* level mapping tables

Shop_V7_V6_Map_Table			
old_tab_name	old_sys_id	new_tab_name	new_sys_id
Shop	400	Shop	500
Shop	200	Shop	600

City_V7_V6_Map_Table			
old_tab_name	old_sys_id	new_tab_name	new_sys_id
Shop	400	City	10
Shop	200	City	20

While executing a query that computes sale of products by shops, for example, the query parser will search the discussed dictionary tables and will construct appropriate partial queries, depending whether *city_name* is stored in the

Shop or *City* level. In this case, partial query result will be annotated with the following metadata information:
Dimension Shop: level detached City
Dimension Shop: source attribute:
Shop.city_name → *City.city_name*

5.2 Heterogeneous fact tables and homogeneous dimension instance structures

Fact tables may differ form version to version with respect to their schemas. Let us consider three cases including table name changing, attribute name changing, and attribute domain changing.

Case 1: Different fact table names. In this case a table in a parent DW version and its corresponding table in a child DW version differ only with respect to their names. The change of a table name is registered in the *Facts_Mappings* data dictionary table, cf. Figure 5. Records in this table map an old fact table (attribute *fm_old_id*) to a new fact table (attribute *fm_new_id*). While parsing a multiversion query the parser searches the content of *Facts_Mappings* and puts a right table name into every partial query. Then, appropriate query result set is annotated with metadata information about changing table name. If for example, table *Sale* is renamed to *Poland_Sale* in version *RV8* then the result set of partial query addressing *RV8* is annotated with:
Table name changing: Sale → *Poland_Sale*

Case 2: Different attribute names. In this case a table in a parent DW version and its corresponding table in a child DW version differ only with respect to their attribute names. The change of an attribute name is registered in the *Attr_Mappings* data dictionary table, similarly as in the above case. The query parser puts appropriate attribute names to appropriate partial queries. Then, results of partial queries are annotated with:
Attribute name changing:
old attribute name → *new attribute name*

Case 3: Different attribute domains. Corresponding attributes of two tables in two adjacent versions may differ with respect to their domains. In order to process a multiversion query and compare partial result sets the values of this attributes have to be mapped to each other. In our approach and prototype system, domain mappings are expressed by means of user defined functions. These functions are registered in attribute mapping records, as discussed earlier, in the *Attr_Mappings* data dictionary table. The name of a forward mapping function, i.e. from a parent to a child DW version is stored as the value of *am_forward_meth_name*, whereas the name of a backward mapping function is stored in *am_backward_meth_name*. As an example let us consider a high school information system with a fact table storing grades of students. In one version grades are represented in the US standard, with values A, B, C, D, and F. Whereas in another version grades are stored in the Polish standard, with values 5, 4, 3, and 2. In such a case, partial results of a user query will be returned, but the integration of this partial results will not be possible unless appropriate grade conversion methods are available in a DW dictionary.

Yet another example may consider changing prices of products from Polish zloty to euro. Appropriately defined conversion method from PLN to EUR and vice versa will allow to compare sales of products.

5.3 Integrating partial queries

Ideally, integration of partial queries is possible if every version of a table, either fact of dimension level, has the same schema. However, in some DW versions an attribute of a fact or dimension level table may exist while in some others may not. Such attributes will be called *missing attributes.* Using missing attributes in the `select`, `where`, `group by`, and `having` SQL clauses causes that integration of partial result sets is not possible.

At the implementation level, the integration is done as follows. Firstly, the query parser finds out a common set of attributes/expressions used in the `select` clause of partial queries. Tables that changed their names and attributes that changed their names and/or domains from version to version are also included and handled, as discussed in the previous section. Secondly, a temporary table is being created for holding partial results. Next, partial queries are executed and all partial result sets being integrated are stored in this temporary table. While loading partial result sets, data are transformed by conversion functions, if needed, in order to form a homogeneous result. Then an integrated result is returned to a user by a query on the temporal table.

Missing attributes in the *select* clause. In order to illustrate this case, let us assume that in January, February, and March DW versions, there is no attribute *total_price* in the *Sale* fact table. The attribute was added in the April version and exists in May and June versions. If a user specified a query computing not only `sum(amount)` but also the `sum (total_price)` of respective products sold by existing shops in the period from 1^{st} January, 2004 until 30^{th} June, 2004, then the query could be answered only in the last three DW versions. In this case the parser module removes the `sum(total_price)` expression from queries passed to versions from January till March. The parser module consults the MVDW data dictionary in order to check schemas of versions of interest. The results which are possible for computation in these versions are then returned to a user. In this case, the second step, i.e. the integration of partial results will not be possible, as data from January, February, and March have different structure than data from April, May, and June.

Missing attributes in the *where*, *group by*, and *having* clauses. Missing attributes used in the the `where`, `group by`, and `having` clauses cause that only those partial queries that address versions having these attributes are executed. In a consequence, only the results of these partial queries will be integrated. Taking into account the above example, if a user limited a query result to only products having `sum(total_price)` grater than 30000, then partial queries would address versions from April until June only.

6. METAMODEL OF A MVDW

The metamodel of a MVDW is shown in Figure 5 using the Oracle notation. The *Versions* dictionary table stores the information about existing DW versions: version identifier, name, parent-child dependencies, validity times, status (whether a version is committed or under development). Every DW version is composed of fact tables (dictionary table *Facts*) and dimensions (dictionary tables *Dimensions* and *Dimensions_Versions*). Dimensions, in turn, have levels represented by dictionary tables *Levels.* Associations between a dimension and its levels are stored in *Dimensions_Levels.*

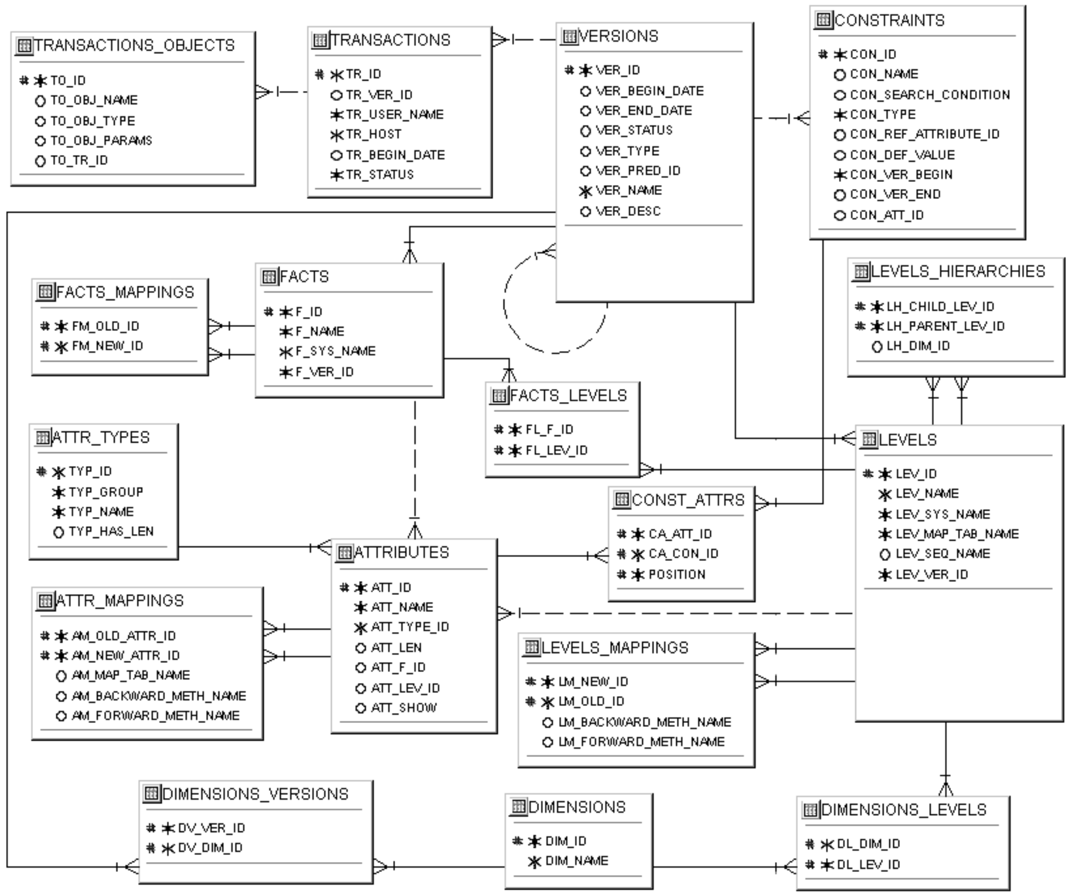


Figure 5: A metamodel of the multiversion data warehouse

Level hierarchies are stored in *Levels_Hierarchies*. Associations between fact tables and dimensions are via levels and are stored in *Facts_Levels*. The *Facts_Mappings* dictionary table is used for storing mappings between a given fact table in DW version V_i and the corresponding fact table in version V_j, directly derived from V_i, as discussed in Section 5.2.

Every fact and level table has the set of its attributes, that are stored in *Attributes* and *Attr_Types*. Every attribute may have integrity constraints defined, that are stored in *Constraints* and *Const_Attrs*. Table *Attr_Mappings* is used for storing mappings between an attribute existing in DW version V_i and the same attribute in a child version V_j, as discussed in Section 5.2. The *Levels_Mappings* table represents mappings between levels in consecutive DW versions, as discussed in Section 5.1. The *Transactions* table stores the information about transactions used for creating new versions of a DW, whereas *Transactions_Objects* stores information about DW objects created, modified, or dropped by transactions.

7. SUMMARY

Handling changes in external data sources, supplying data to a data warehouse, and applying the changes to a DW became important research and technological issues. Structural changes to a DW schema applied inappropriately may result in wrong analytical results. Most of commercial DW systems existing on the market have static structure of their schemas and relationships between data. In a consequence, they are not well suited for handling of any changes that occur in a real world. Moreover, the existing systems do no support the creation of alternative business scenarios for the purpose of the what-if analysis. Research prototypes and solutions to this problem are mainly based on temporal extensions that limit their use.

Our approach to this problem is based on a multiversion data warehouse, where a DW version represents the structure and content of a DW at a certain time period. In order to be able to analyze the content of a multiversion DW, a query language was extended. The presented approach and prototype MVDW system allows to query several DW versions, present their partial results annotated with version and metadata information and, if possible, it allows to integrate partial results into a single homogeneous result set. The metadata information allows to appropriately analyze the results under schema changes and dimension instance structure changes in DW versions.

The current implementation of our MVDW query interface has several limitations (cf. Section 4.3) that are going to be removed in the next release of the system. Future work will also concentrate on designing and implementing a buffering layer between a MVDW and external data sources for automatic discovery of structural changes in EDSs, that have an impact on a DW schema, and automatically applying those changes to a MVDW.

8. REFERENCES

[1] Balmin, A., Papadimitriou, T., Papakonstanitnou, Y.: Hypothetical Queries in an OLAP Environment. Proc. of the VLDB Conf., Egypt, 2000

[2] Bellahsene, Z.: View Adaptation in Data Warehousing Systems. Proc. of the DEXA Conf., 1998

[3] Bebel B., Eder J., Konicilia C., Morzy T., Wrembel R.: Creation and Management of Versions in Multiversion Data Warehouse. Proc. of the ACM Symposium on Applied Computing (SAC'2004), Cyprus, 2004

[4] Bebel B., Królikowski Z., Morzy T., Wrembel R.: Transaction Concepts for Supporting Changes in Data Warehouses. Proc. of the Int. Conf. on Enterprise Information Systems, Portugal, 2004

[5] Blaschka, M. Sapia, C., Hofling, G.: On Schema Evolution in Multidimensional Databases. Proc. of the DaWak99 Conference, Italy, 1999

[6] Body, M., Miquel, M., Bédard, Y., Tchounikine A.: A Multidimensional and Multiversion Structure for OLAP Applications. Proc. of the DOLAP'2002 Conf., USA, 2002

[7] Chamoni, P., Stock, S.: Temporal Structures in Data Warehousing. Proc. of the DaWaK99, Italy, 1999

[8] Eder, J., Koncilia, C.: Changes of Dimension Data in Temporal Data Warehouses. Proc. of the DaWaK 2001 Conference, Germany, 2001

[9] Eder, J., Koncilia, C., Morzy, T.: The COMET Metamodel for Temporal Data Warehouses. Proc. of the 14th CAISE02 Conference, Canada, 2002

[10] Gyssens M., Lakshmanan L.V.S.: A Foundation for Multi-Dimensional Databases. Proc. of the 23rd VLDB Conference, Grece, 1997

[11] Hurtado, C.A., Mendelzon, A.O.: Vaisman, A.A.: Maintaining Data Cubes under Dimension Updates. Proc. of the ICDE Conference, Australia, 1999

[12] Hurtado, C.A., Mendelzon, A.O.: Vaisman, A.A.: Updating OLAP Dimensions. Proc. of the DOLAP Conference, 1999

[13] Jarke, M., Lenzerini, M., Vassiliou, Y., Vassiliadis, P.: Fundamentals of Data Warehouses. Springer-Verlag, 2000, ISBN 3-540-65365-1

[14] Kaas C.E., Pedersen T.B., Rasmussen B.D.: Schema Evolution for Stars and Snowflakes. Proc. of the Intern. Conf. on Enterprise Information Systems (ICEIS2004), Portugal, 2004

[15] Kang, H.G., Chung, C.W.: Exploiting Versions for On–line Data Warehouse Maintenance in MOLAP Servers. Proc. of the VLDB Conference, China, 2002

[16] Koeller, A., Rundensteiner, E.A., Hachem, N.: Integrating the Rewriting and Ranking Phases of View Synchronization. Proc. of the DOLAP98 Workshop, USA, 1998

[17] Kulkarni, S., Mohania, M.: Concurrent Maintenance of Views Using Multiple Versions. Proc. of the Intern. Database Engineering and Application Symposium, 1999

[18] Letz C., Henn E.T., Vossen G.: Consistency in Data Warehouse Dimensions. Proc. of the Intern. Database Engineering and Applications Symposium (IDEAS'02), 2002

[19] Mendelzon, A.O., Vaisman, A.A.: Temporal Queries in OLAP. Proc. of the VLDB Conference, Egypt, 2000

[20] Morzy, T., Wrembel, R.: Modeling a Multiversion Data Warehouse: A Formal Approach. Proc. of the Int. Conf. on Enterprise Information Systems, France, 2003

[21] Quass, D., Widom, J.: On–Line Warehouse View Maintenance. Proc. of the SIGMOD Conference, 1997

[22] Rundensteiner E., Koeller A., and Zhang X.: Maintaining Data Warehouses over Changing Information Sources. Communications of the ACM, vol. 43, No. 6, 2000

[23] Vaisman A.A., Mendelzon A.O., Ruaro W., Cymerman S.G.: Supporting Dimension Updates in an OLAP Server. Proc. of the CAISE02 Conference, Canada, 2002

[24] Oracle Database. Data Warehousing Guide. 10g Release 1.