Schema Evolution in Multiversion Data Warehouses

Waqas Ahmed, CODE WIT, Université libre de Bruxelles, Belgium
Esteban Zimányi, CODE WIT, Université libre de Bruxelles, Belgium
https://orcid.org/0000-0003-1843-5099

Alejandro A. Vaisman, Department of Information Engineering, Instituto Tecnológico de Buenos Aires, Argentina
https://orcid.org/0000-0002-3945-4187

Robert Wrembel, Poznan University of Technology, Poland
https://orcid.org/0000-0001-6037-5718

INTRODUCTION

A Data Warehouse (DW) integrates data coming from different sources (Vaisman & Zimányi, 2014) and supply it to various systems, including Business Intelligence (BI) applications. Data warehouses change in their content and schema. Typically, content changes are due to routine business operations or the correction of existing data. An example of a content change is a modification in the price of a product.

In practice, changes to a DW schema result from (1) the evolution of external data sources, (2) changes of the real-world represented by a DW, (3) new user requirements, and (4) the creation of simulation environments to list the most common causes. An example of a schema change is a change in the geographical hierarchy of a sales network. As reported in (Moon, Curino, Deutsch, Hou, & Zaniolo, 2008; Qiu, Li, & Su, 2013; Sjøberg, 1993; Vassiliadis, Zarras, & Skoulis, 2017), the schemas of data sources change frequently. For example, the Wikipedia schema changed on average every 9-10 days during the 4.5 years of its lifetime. As a result of the changes in data sources, the content and schema of the related DWs must also change. Real-world examples of scenarios leading to changes in DWs can be found in (Eder, Koncilia, & Kogler, 2002; Rundensteiner, Koeller, & Zhang, 2000).

A DW should keep track of the evolution of its content and schema to reconstruct the state of the world under consideration at any instant without losing data. A temporal data warehouse (TDW) (Golfarelli & Rizzi, 2009) keeps track of the evolution of its contents whereas, a multiversion data warehouse (MVDW), based on multischema data management principles (Roddick, 1995; Herrmann, Voigt, Pedersen, & Lehner, 2018), handles content and schema changes by creating multiple and persistent DW versions.

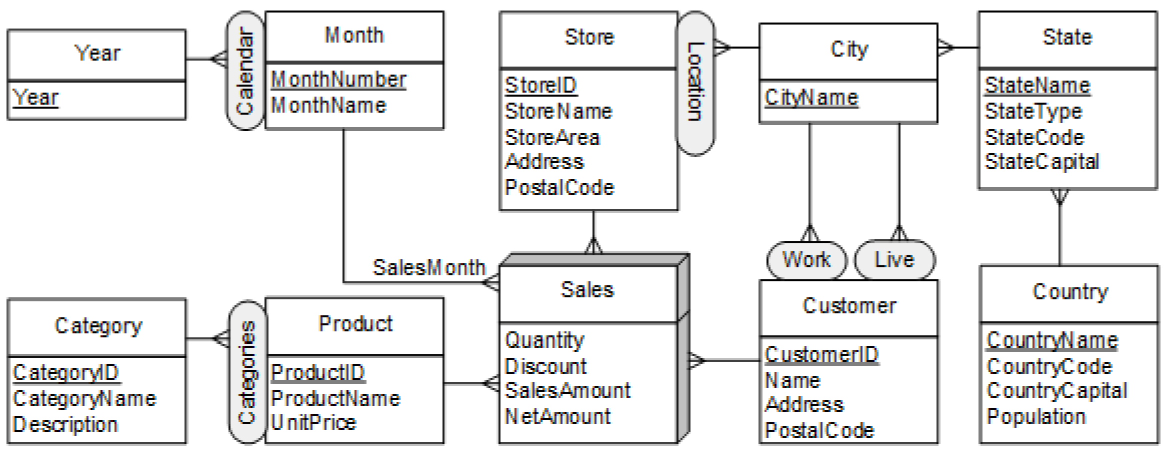
Even though version management in databases has been researched for over 30 years (in the context of object databases, relational databases, data warehouses, XML databases), it is still an active research field. This research is regaining its importance in the context of NoSQL storage and data lakes. Support for version management was explicitly stated as a requirement for data lakes management systems (Nargesian, Zhu, J. Miller, Pu, & Patricia C. , 2019).

The temporal DW-based approaches profit from the support of various temporal query languages to analyze changing data and from the existence of multiple index structures. Such approaches are suitable for representing historical data versions and not for representing and managing schema changes. On the other hand, the MV approaches allow managing both data and schema changes. However, the implementation of such approaches is more complicated. Moreover, they possess the limited capabilities of querying DW versions.

Multidimensional Data Model

Usually, data in a DW is represented using the multidimensional model (MD), which stores data as a collection of *facts, measures, dimensions, levels, and hierarchies*. These notions are informally introduced next through a running example that represents sales in a fictitious company. The initial DW version is depicted in Figure 1, using the MultiDim (Vaisman & Zimányi, 2014) notation.

Figure 1 The initial DW version V_1 to analyze the sales of a company.



In a MD model, data are perceived in an *n*-dimensional space. In this space, a *fact* is a subject of interest. Each observation in a fact is called a *fact member*. *Measures* are numerical quantities that quantify a fact. *Dimensions* provide context to facts. For example, sales events can be perceived in a three-dimensional Sales fact, contextualized by dimensions Product, SalesDate, Store, and quantified by measure Quantity.

Levels are described by attributes and provide dimension values. For example, level Product provides all possible values for dimension Product. Instances of a level are called *level members*. A DW may contain multiple levels, and multiple facts may share these levels. A level is connected to a fact if it provides values for any dimension in it. Such a level determines that dimension's granularity, which is the level of details at which measures are recorded. For example, in the fact Sales in Figure 1, values for dimension SalesDate come from level Month; therefore, the dimension's granularity is at the Month level. A level may provide values for more than one dimension, and such dimensions are called *role-playing* dimensions.

The relationship between levels is called an *aggregation relationship*, which associates the members of a parent level and a child level. The latter is the level defined at a finer granularity in the relationship. Further, the *cardinality* of an aggregation relationship indicates how level members relate to each other. Analogously, the cardinality of the relationship between a fact and a level indicates how fact members are associated with level members.

Like conceptual modeling, cardinalities can be one-to-one (1-1), many-to-one (m-1), and many-to-many (m-m). Furthermore, the cardinality can be optional (denoted *0*) for any of the two participating entities of a relationship, meaning that the participation of members of an entity is not mandatory in the relationship. For example, the cardinality between level Product and fact Sales is many-to-one, meaning that a sales transaction contains one product, and a product member may appear in multiple sales transactions. A *roll-up hierarchy* is a collection of logically related aggregation relationships that allows aggregating measure values to a coarser level of detail from values at a finer level of detail.

Often, the MD data is analyzed using a sequence of the so-called OLAP operators. These operators include *roll-up, dice*, *project, rename*, and *drill-across*. The *roll-up* operation transforms data from a lower level of details to a higher level of detail, for example, aggregating the daily sales to monthly sales. The *dice* operation filters out specific data from the DW; for example, viewing the sales made in Belgium only. The *project* operation reduces the dimensionality of a fact; for example, eliminating store dimension from fact Sales. The *rename* operation renames a dimension in a fact; for example, changing dimension name SalesDate to OrderDate. Finally, the *drill-across* operation correlates data from another fact; for example, merging the fact Sales with SalesForecast to analyze the actual and forecast yearly sales together.

Running Example

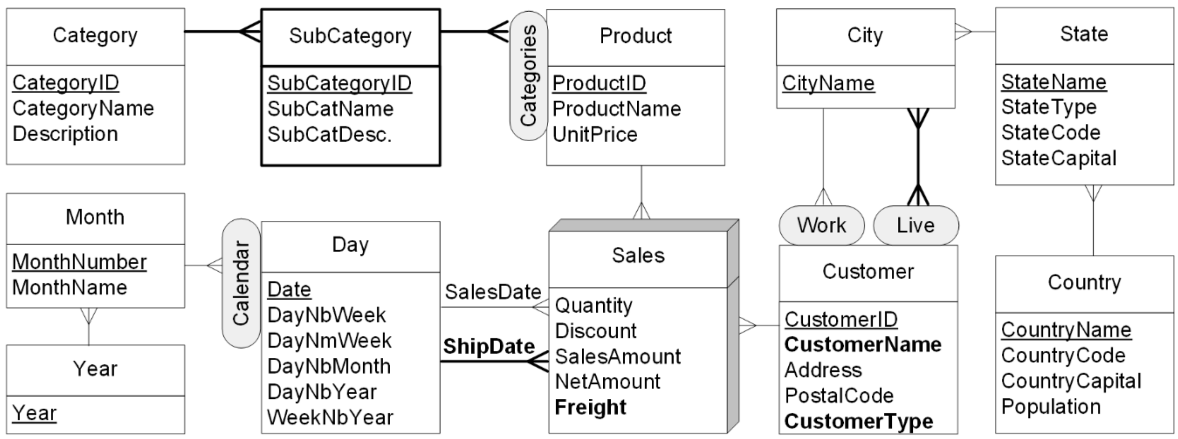
Consider the following changes applied to the initial version V_1 from Figure 1, which was created at the instant t_0.

- C1. Level Store is deleted from the DW; thus, dimension Store is removed from fact Sales.
- C2. A new level SubCategory is added, and aggregation relationships between the new level and Product and Category are defined.
- C3. Level Day is added, and it is also linked to fact Sales. As a result, the granularity of dimension SalesDate is changed from level Month to Day. Moreover, an aggregation relationship Day_Month is created between levels Day and Month.
- C4. A role-playing dimension ShipDate is added to the fact.
- C5. In level Customer, attribute Name is renamed to CustomerName, and a new attribute CustomerType is added.
- C6. The cardinality of aggregation relationship Customer_City_Lives between levels Customer and City is changed from m-1 to m-m.
- C7. Finally, a new measure Freight is added to fact Sales.

Due to schema changes, the data corresponding to new schema elements becomes available, and data corresponding to the existing elements becomes unavailable. For example, after the change denoted C1, the information about stores where sales occur, is not available for the new fact members. After C3, the information about daily sales becomes available. One way to handle these changes is through the support of *schema evolution,* in which the existing data is conformed and exported to the new schema. However, this may lead to the loss of some information. For example, if dimension Store from fact Sales is deleted in the new schema, then the store information of the existing fact members is lost. Furthermore, all existing applications and reports must be adapted to consume data from the DW using the new schema.

An alternative approach to handle schema changes is using *schema versioning,* where changes are handled by creating and maintaining DW versions. The above modifications at t_4 in DW version V_1 result in creating another DW version V_2 (shown in Figure 2).

Figure 2 DW version V_2 after applying schema changes at t_4.



Research Problem

Given that a DW is, by definition, historical and time-varying, a MVDW must (1) retain all the data loaded into it throughout its lifespan; (2) allow accessing all the stored data using any schema version; and (3) enable OLAP operations producing semantically correct results. The first requirement is straightforward, implying that no data will ever be deleted from a DW. However, the second and third requirements are not trivial and need the following considerations.

First, data can be stored at various granularities using multiple schemas; hence, it must be transformed to conform to the queried schema. For example, C3 resulted in the change of the granularity of dimension SalesDate from Month to Day. Subsequently, from t_4 onwards, all fact members will be daily sales whereas, the fact members before t_4 were the monthly sales. To access all sales using schema version active at t_1, the daily sales after t_4 need to be converted into monthly sales. Conversely, to access the sales using schema active after t_4, the monthly sales recorded before t_4 must be disaggregated to daily sales.

Second, data of a schema element in the queried schema may not be present in at least one of the DW schemas, leading to schema incompatibility and missing information. For instance, the store information for the sales recorded after t_4 could be unavailable; therefore, the breakdown of *sales amount per store* could not add up to the *total sales amount*.

Contributions and Paper Organization

In our previous work (Ahmed, Zimányi, Vaisman, & Wrembel, 2020), we proposed a MD model to manage the content changes. This paper extends the MD model with multiversion capability to offer the advantages of both approaches. On the one hand, the temporal MD model provides an easier-to-implement approach to manage content changes. On the other hand, handling content changes in MVDWs is complicated; however, they can manage schema changes. Thus, the temporal and MV MD models complement each other and can be used to manage content and schema changes independently.

This paper proses a MVDW which has the following features.

- The MVDW stores only once all the members of all versions of a level and fact to avoid data deduplication. These stored members are then shared among various level and fact versions via data mappings. Opposite to various database version management approaches, which require bidirectional mappings between consecutive schema versions, only unidirectional mappings are needed. This approach makes version management and querying a MVDW more efficient.
- Schema changes in MVDW can be carried out via schema modification operators (SMOs). The semantics of the included SMOs are given for a MD model, which are independent of the underlying implementation. For example, if a MVDW is implemented on a relational database, these MD SMOs can be translated into relational SMOs (Curino, Moon, Ham, & Zaniolo, 2009; Herrmann, Voigt, Pedersen, & Lehner, 2018). Defining SMOs for the MD model makes the schema evolution concise and elegant. For instance, in a relational DW, the deleteRelationship SMO is a concise representation of deleting a foreign key constraint and a column from a table. Furthermore, a user can derive new schema versions by operating over a more familiar model without interacting with the underlying low-level structures.
- The MVDW can deal with content and schema changes separately. The DW versions are created only upon schema changes, and the content changes can be managed using temporal DWs. In this way, the user can take advantage of temporal DW features to store and query time-evolving data and use MVDW to manage the schema changes.
- Each DW version behaves as a complete DW with all the data stored using all versions; therefore, traditional OLAP operators can be used to query data from any DW version.

The rest of the paper is organized as follows. It begins with an intuitive explanation of the proposed MVDW and the semantics of the included SMOs. Then, the constructs of a MVDW are formalized, and OLAP operations on it are shown. After this, the paper shows how the formal constructs of the MVDW, including the data consistency constraints, can be mapped into a relational schema. Also, with the help of example queries, it is shown how the model's OLAP operations can be implemented in the standard SQL. A study of the related work follows, and finally, the conclusions and future research directions are given.

MULTIVERSION DATA WAREHOUSE

This section gives an intuitive explanation of the components of a MVDW. A MVDW consists of DW versions. Each DW version comprises a schema version, which defines the structure of data, and an instance version, which includes data that conform to a given schema version.

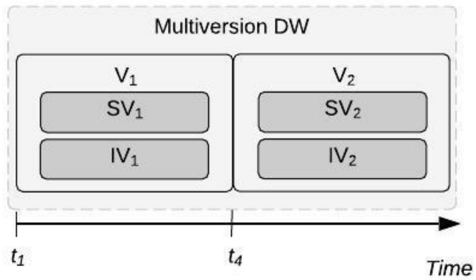
Linear of branched versioning model (Wrembel & Bebel, Metadata management in a multiversion data warehouse, 2005) can be used to create DW versions. In the *linear versioning* model, the DW versions are linearly ordered to the time they are created. Also, a DW version is derived from the latest version only, which is also used to store the new data. Moreover, at a given instant, the version that is used to store or access data is called the *active version*. In the *branched versioning* model, more than one schema version can be derived from the latest version. The linear versioning model is straightforward and captures real-world business changes. The branched versioning model requires more maintenance but allows simulating alternative business scenarios. This paper considers the linear versioning model; however, the MVDW can be generalized to enable branched versioning. The history of multiple DW versions is stored as a DW *version derivation graph*, and for the example MVDW, it is shown in and create new versions of either DW or its elements. For instance, the schema changes in level Customer create a new version of the level. The initial and new versions of the level are shown in Figure 4 and Figure 5, respectively. Furthermore, the new DW version V_2 includes Customer_{v2}. A functional description of the SMOs is given in When creating a new element version, an SMO modifies the corresponding metaelement if needed, creates a new element version and a mapping between it and the corresponding metaelement. For example, addAttribute(Customer_{v1}, CustomerType: String) add a new attribute customerType to the metalevel Customer_{meta}, and creates a new level version Customer_{v2}, and created and mapping between Customer_{v2} to Customer_{meta}.

A fact version comprises dimensions and measures, each of which has a corresponding dimension and measure in the metafact. A metafact is composed of the union of all dimensions and measures created in all versions of a fact. The schema of a metalevel and metafact is append-only; that is, because of schema changes, no element is deleted from it. In this way, all existing data are preserved.

The additional elements of a MVDW are aggregation relationships and hierarchies, which are composed of their versions. An aggregation relationship version associates members of two levels. A hierarchy version is composed of logically ordered aggregation relationship versions. Figure 7 schematically shows all components of a MVDW.. Multiple SMOs can be grouped in a transaction to derive a new element version. All SMOs are information preserving; that is, no information is lost by applying any SMO. All versions of an element form an element version derivation graph independent of the DW version derivation graph and have the first element versions at the root.

.

Figure 3 Version derivation graph of the DW versions given in Figures 1 and 2. Version V_1 and V_2 consist of schema versions SV_1 and SV_2, and instance versions IV_1 and IV_2, respectively.



Schema changes in a DW version trigger the creation of new versions. These changes may affect the overall schema, such as adding level SubCategory and deleting level Store from the initial DW version V_1, or may affect an element within a version, such as adding attribute CustomerType to level Customer.

Schema modification operators (SMOs) are used to carry out such changes and create new versions of either DW or its elements. For instance, the schema changes in level Customer create a new version of the level. The initial and new versions of the level are shown in Figure 4 and Figure 5, respectively. Furthermore, the new DW version V_2 includes $Customer_{v2}$. A functional description of the SMOs is given in When creating a new element version, an SMO modifies the corresponding metaelement if needed, creates a new element version and a mapping between it and the corresponding metaelement. For example, addAttribute($Customer_{v1}$, CustomerType: String) add a new attribute customerType to the metalevel $Customer_{meta}$, and creates a new level version $Customer_{v2}$, and created and mapping between $Customer_{v2}$ to $Customer_{meta}$.

A fact version comprises dimensions and measures, each of which has a corresponding dimension and measure in the metafact. A metafact is composed of the union of all dimensions and measures created in all versions of a fact. The schema of a metalevel and metafact is append-only; that is, because of schema changes, no element is deleted from it. In this way, all existing data are preserved.

The additional elements of a MVDW are aggregation relationships and hierarchies, which are composed of their versions. An aggregation relationship version associates members of two levels. A hierarchy version is composed of logically ordered aggregation relationship versions. Figure 7 schematically shows all components of a MVDW.. Multiple SMOs can be grouped in a transaction to derive a new element version. All SMOs are information preserving; that is, no information is lost by applying any SMO. All versions of an element form an element version derivation graph independent of the DW version derivation graph and have the first element versions at the root.

The elements of a DW version are *level versions*, *fact versions*, *aggregation relationship versions*, and *hierarchy versions*. These elements behave like regular levels, facts, aggregation relationships, and hierarchies, respectively. A DW version may have only one version of any element. Also, the element versions that do not evolve between DW versions are shared among them.

A level and a fact in the MVDW consist of its versions and a *metalevel* and a *metafact*, respectively. The meta elements allow storing element members only once, which then can be shared among multiple element versions. In this way, element versions serve as an interface to access or store data into the corresponding metaelement.

A *level version* is composed of attributes, and each attribute has a corresponding attribute in the metalevel. The metalevel $\text{Customer}_{\text{meta}}$ is shown in Figure 6, and it consists of all attributes in both versions of the level.

Figure 4 Level version $\text{Customer}_{\text{v1}}$ *after* t_4.

CustomerID	Name	Address	PostalCode
c_1	Galaxy Corp	Rue Fosses 215	1050
…	…	…	…
c_2	Ostato	BD Basse 428	1070
…	…	…	…

Figure 5 Level version $\text{Customer}_{\text{v2}}$ *after* t_4.

CustomerID	CustomerName	Address	PostalCode	CustomerType
c_1	Galaxy Corp	Rue Fosses 215	1050	Default
…	…	…	…	…
c_2	Ostato	BD Basse 428	1070	Individual
…	…	…	…	…

Figure 6 Metalevel $\text{Customer}_{\text{meta}}$ *after* t_4.

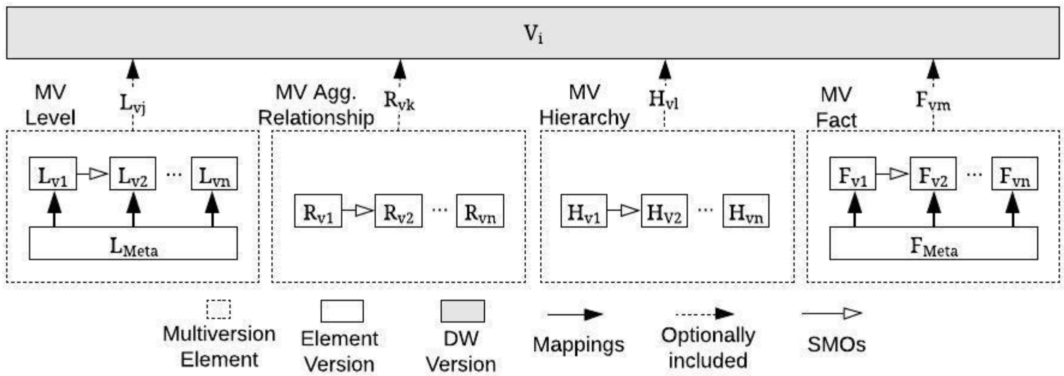
CustomerID	Name	Address	PostalCode	CustomerType
c_1	Galaxy Corp	Rue Fosses 215	1050	Default
…	…	…	…	…
c_2	Ostato	BD Basse 428	1070	Individual
…	…	…	…	…

When creating a new element version, an SMO modifies the corresponding metaelement if needed, creates a new element version and a mapping between it and the corresponding metaelement. For example, addAttribute($\text{Customer}_{\text{v1}}$, CustomerType: String) add a new attribute customerType to the metalevel $\text{Customer}_{\text{meta}}$, and creates a new level version $\text{Customer}_{\text{v2}}$, and created and mapping between $\text{Customer}_{\text{v2}}$ to $\text{Customer}_{\text{meta}}$.

A fact version comprises dimensions and measures, each of which has a corresponding dimension and measure in the metafact. A metafact is composed of the union of all dimensions and measures created in all versions of a fact. The schema of a metalevel and metafact is append-only; that is, because of schema changes, no element is deleted from it. In this way, all existing data are preserved.

The additional elements of a MVDW are aggregation relationships and hierarchies, which are composed of their versions. An aggregation relationship version associates members of two levels. A hierarchy version is composed of logically ordered aggregation relationship versions. Figure 7 schematically shows all components of a MVDW.

Figure 7 Schematic representation of a MVDW.



When accessing all MVDW data using a schema version, some data may not conform to the active schema. A *nonconformity* arises when the data of an element in the active schema may be unavailable or indirectly available in any of the other schema versions. For instance, if all customers are accessed using $Customer_{v2,}$ then the value of CustomerType is unavailable for members stored using $Customer_{v1}$. However, the problem of nonconformity does not arise when all customers are accessed using $Customer_{v1}$.

Coercion functions (Merlo, Bertino, Ferrari, & Guerrini, 1999; Malinowski & Zimányi, 2008) ensure the data conformance among schema versions. A *coercion function* gives the default value of an attribute, dimension, or measure. It can also provide a default parent for a child member in a child-parent relationship. For instance, the function defCustType can set the CustomerType to a default value for members whose attribute value is unavailable. A brief description of the nonconformity arising because of each schema change is also given in When creating a new element version, an SMO modifies the corresponding metaelement if needed, creates a new element version and a mapping between it and the corresponding metaelement. For example, addAttribute($Customer_{v1}$, CustomerType: String) add a new attribute customerType to the metalevel $Customer_{meta}$, and creates a new level version $Customer_{v2}$, and created and mapping between $Customer_{v2}$ to $Customer_{meta.}$

A fact version comprises dimensions and measures, each of which has a corresponding dimension and measure in the metafact. A metafact is composed of the union of all dimensions and measures created in all versions of a fact. The schema of a metalevel and metafact is append-only; that is, because of schema changes, no element is deleted from it. In this way, all existing data are preserved.

The additional elements of a MVDW are aggregation relationships and hierarchies, which are composed of their versions. An aggregation relationship version associates members of two levels. A hierarchy version is composed of logically ordered aggregation relationship versions. Figure 7 schematically shows all components of a MVDW..

The difference in the granularity of a dimension also introduces a nonconformity. For example, after C3, version $Sales_{v2}$ of fact Sales is created, which stores the daily sales. Since the initial version $Sales_{v1}$ was keeping monthly sales, the two versions' sales do not conform to each other.

Therefore, when accessing all sales using $Sales_{v1}$, daily sales need to be converted to monthly sales and vice versa. To handle C3, a new dimension SalesDate is added to the metafact shown in Figure 8, and it records the day of sales. The coercion function dayMonth can obtain the month each day belongs to using the aggregation relationship Day_Month as shown in Figure 9 for fact members added using $Sales_{v2}$ (gray shaded).

Table 1 Functional description of the schema modification operations (SMOs).

Schema change	**Nonconformity**	**Semantics of SMO**
Add level/fact	None - Only the queries written against the new schema will mention new level/-fact, thus there is no impact on the existing queries.	Create a new MV level/fact.
Delete level/fact	The deleted level/fact is still available in the existing versions. However, the new data will not be loaded for it.	Do nothing
Add attribute/ measure	The existing level or fact members will not have values for the added attribute/measure, respectively.	Add a new attribute/measure to the metalevel/metafact and create a new level/fact version with the new attribute included.
Delete attribute/ measure	The new level or fact members will not have values for the deleted attribute/measure, respectively.	Create a new version of the level/fact with the new attribute excluded.
Rename attribute/ measure	None - However, an alias mapping is required to map the renamed attribute/measure to an attribute/measure in the metalevel or metafact, respectively.	Create a new level/fact version with the renamed attribute included and create a a mapping between renamed attribute/measure and an attribute/measure in the metalevel/metafact.
Change attribute/ measure domain to specific	It may not be possible to convert the existing values to the new type, e.g., string to int. Also, there may be a loss of information for the existing measure values, e.g., converting float to int.	Add a new attribute/measure with the changed type in the meatalevel/metafact and convert the values for the existing members to the new type.
Change attribute/ measure domain to generic	The attribute/measure values for the new level/fact members may not be available in the existing schema as they may not be convertible to the previous specific type, e.g., change from int to string.	Add a new attribute/measure with the changed type in the meatalevel/metafact and convert the values for the existing members to the new type.
Add relationship	The existing child members will become orphans unless their parents are explicitly specified.	Create a new relationship between parent and child levels and link the orphan child members to the default parent member.
Delete relationship	The new child level members will become orphan.	Do nothing
Change cardinality (m-m to 1-m)	The child-parent relationship between existing members may violate the cardinality constraint in the new schema.	Create a new relationship version. Also, use a function to convert m-m to 1-m for the existing members. The new members must adhere to the constraint by default.
Change cardinality (1-m to m-m)	The child-parent relationship between the new members may violate the constraint in the previous schema.	Create a new relationship version. Also, use a function to convert m-m to 1-m for the new members.
Change cardinality (make optional)	The new members may violate the constraint in the previous schema.	Create a new relationship version and link the orphan members to the default parent member.
Change cardinality (make mandatory)	The existing members may violate the constraint in the new schema.	Link the orphan members to the default parent member.
Add/delete level to/from hierarchy	None	Create a new hierarchy version with the level added/deleted in it.
Add dimension to fact	The existing fact members will not have a dimension value for the newly added dimension thus may not be available for OLAP operations involving this recently added dimension.	Create a new fact version with the new dimension and obtain the dimension values for the existing fact members as per the coercion function.
Delete dimension from fact	The new fact members will not have dimension value for the existing dimension. They thus may not be available for OLAP operations involving the deleted dimension.	Create a new fact version with the deleted dimension excluded and obtain the dimension values for the new fact members as per the coercion function.

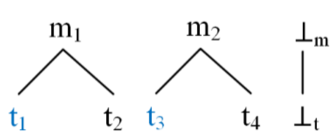
Make granularity finer/coarse	The semantics of the data may become different in two versions, e.g., daily vs monthly sales.	Treat as adding a new dimension to a fact.

Similarly, the function monthDay gives the value of SalesDate for fact members added using Sales_{v1}. The function can map each month to a default day, for example, the first day of the month. The default day of each month is shown in blue in Figure 9. In Figure 8, the values obtained from coercion functions are bounded by the blue boxes.

Figure 8 Fact Sales_{meta} *where missing values are obtained from coercion functions.*

SalesKey	SalesMonth	SalesDate	Quantity
1	m_1	t_1	10
2	m_1	t_1	20
3	m_2	t_2	10
4	m_2	t_2	10
5	m_1	t_1	2
6	m_1	t_2	3
7	m_2	t_3	5

Figure 9 Aggregation relationship Day_Month



The change of granularity from finer to coarser can also be handled in the same way as above. As an example, if the granularity of SalesDate is changed from the Day to Month level, then the new and existing fact members will not have values for SalesDate and SalesMonth, respectively. However, as in the case of change of granularity from Month to Day level, coercion functions monthDay and dayMonth can provide the values for SalesDate and SalesMonth, respectively.

A FORMAL MULTIVERSION DATA WAREHOUSE MODEL

This section formalizes the MVDW given in the previous section and shows how OLAP operations can be performed on it.

A Multidimensional Multiversion Data Warehouse

In what follows, $\beta = \{Boolean, Integer, Real, String\}$ is a set of base types. Furthermore, the domain of a base type $b \in \beta$ is denoted $dom(b)$, and it is extended with a special value ⊥. Also, $Card = \{01-01, 01-1, 1-01, 1-1, 0m-01, 0m-1, m-01, m-1, 0m-0m, 0m-m, m-0m, m-m\}$ is a set of cardinality constraints whose elements follow the usual semantics as in conceptual modelling.

Definition 1 (Coercion function). A coercion function f is defined as $f: dom(b_1) \times \ldots \times dom(b_n) \rightarrow dom(b)$, such that b and $b_i \in \beta, i = 1, \ldots, n$. ■

Before the definition of a MVDW could be given, it is necessary to define some concepts informally introduced in the previous section formally.

Definition 2 (Level version schema). The schema of a level version L_v is denoted as $L_v(A_k: b_k, A_1: b_1, \ldots, A_n: b_n)$, where, L_v is the level version name, A_k is the name of the key attribute with base type $b_k \in \beta$, and each $A_i, i = 1, \ldots, n$, is an attribute name and it has a base type $b_i \in \beta$. All attribute names are unique in L_v, and the function $attribNames: L_v \rightarrow \{A_1, \ldots, A_n\}$ gives a set of all attribute names of level version L_v. ■

Definition 3 (Level version instance). An instance $[\![L_v]\!]$ of level version L_v from Def. 2 is a set of level members defined by $[\![L_v]\!] \subset \{k \times a_1 \times \ldots \times a_n \mid k \in dom(b_k) \wedge a_i \in dom(b_i), i = 1, \ldots, n\} \cup \{\bar{l}\}$ where $\bar{l}$ is the default member of L_v. The set of the key values of $[\![L_v]\!]$ is denoted by $[\![L_v]\!]_\kappa = \{l.k \mid l \in [\![L_v]\!]\}$. ■

Definition 4 (Metalevel schema). The schema of a metalevel L_{meta} is $L_{meta}\left(A_k: b_k, \left(A_1: b_1, f_{L_1}\right), \ldots, \left(A_n: b_n, f_{L_n}\right)\right)$, where L_{meta} is the metalevel name, A_k is the name of the key attribute, and it has a base type $b_k \in \beta$, each $A_i, i = 1, \ldots, n$, is an attribute name and it has a base type $b_i \in \beta$, and f_{L_i} is a coercion function as defined in Def. 1, and $Range\left(f_{L_i}\right) = dom(b_i)$. All attribute names are unique in L_{meta}, and the function $attribNames: L_{meta} \rightarrow \{A_1, \ldots, A_n\}$ gives a set of all attribute names of L_{meta}. ■

Definition 5 (Metalevel instance). The instance $[\![L_{meta}]\!]$ of metalevel L_{meta} from Def. 4 is a set of level members defined by $[\![L_{meta}]\!] = [\![\widehat{L_{meta}}]\!] \cup \{\bar{l}\}$, where $\bar{l}$ is the default member of L_{meta}, and $[\![\widehat{L_{meta}}]\!] \subset \{k \times g(a_1) \times \ldots \times g(a_n) \mid k \in dom(b_k) \wedge a_i \in dom(b_i), i = 1, \ldots, n\}$, such that

$$g(a_i) = \begin{cases} f_{L_i}(x_1, \ldots, x_p), & if\ a_i = \perp \\ a_i, & otherwise \end{cases}$$

Further, $[\![L_{meta}]\!]$ satisfies the key constraint, that is, $\forall l_1, l_2 \in [\![L_{meta}]\!], l_1 \neq l_2 \Rightarrow l_1.k \neq l_2.k$. The set of the key values of $[\![L_{meta}]\!]$ is denoted by $[\![L_{meta}]\!]_\kappa = \{l.k \mid l \in [\![L_{meta}]\!]\}$. ■

Definition 6 (Level Schema). The schema of a level is $L(L_{meta}, \mathcal{L}_v, \chi_L)$, where L is the name of the level, L_{meta} is a metalevel as in Def. 4, $\mathcal{L}_v = \{L_{v_1}, \ldots, L_{v_n}\}$ is a set of level versions and a level version $L_v \in \mathcal{L}_v$ (given in Def. 2), and $\chi_L = \{\lambda_1, \ldots, \lambda_n\}$ is a set of functions such that $\lambda_j: attribNames\left(L_{v_j}\right) \rightarrow attribNames(L_{meta}), j = 1, \ldots, n$, is a total function. Furthermore, all level versions in L include the same key attribute A_k. ■

Example 1. At t_4, the schema of level Customer is Customer(Customer$_{meta}$, $\mathcal{L}_v$={Customer$_{v1}$, Customer$_{v2}$}, $\chi_L = \{\lambda_1, \lambda_2\}$). The schema of Customer$_{meta}$ is Customer$_{meta}$(CustomerID: *Integer*, (Name: *String*, f_{Name}), (Address: *String*, $f_{Address}$) , (PostalCode:*Integer*, $f_{postCode}$), (CustomerType:*String*, $f_{custType}$)), and CustomerID is the name of the key attribute. The schema of Customer$_{v1}$ is Customer$_{v1}$ (CustomerID: *Integer*, Name: *String*, Address:*String*, PostalCode:*Integer*), and the schema of Customer$_{v2}$ is Customer$_{v2}$ (CustomerID: *Integer*, CustomerName: *String*, Address:String, PostalCode:*Integer*, CustomerType:*Integer*). The function λ_2 maps the attributes of Customer$_{v2}$ to attributes of Customer$_{meta}$ as follows: λ_2 ={(CustomerID, CustomerID), (CustomerName, Name), (Address, Address), (PostalCode, PostalCode), (CustomerType, CustomerType)}.

Definition 7 (Level instance). The instance $[\![L]\!]$ of a level L from Def.6 consists of the instance of its metalevel $[\![L_{meta}]\!]$ and the instance $[\![L_v]\!]$ of each of its level versions $L_v \in \mathcal{L}_v$. The instances $[\![L_{meta}]\!]$ and $[\![L_v]\!]$ are as defined in Def. 3 and 5, respectively. Additionally, each level member in $[\![L_{meta}]\!]$ is present in $[\![L_v]\!]$, that is, $[\![L_{meta}]\!]_\kappa = [\![L_v]\!]_\kappa$. Furthermore, $\forall l \in [\![L_v]\!], \exists l' \in [\![L_{meta}]\!]$, such that $l.A = l'.A'$, such that $(A, A') \in \lambda_v$. ■

Example 2. The level instance ⟦Customer⟧ consists of the instance of metalevel $\text{Customer}_{\text{meta}}$ and its version instances $[\![\text{Customer}_{\text{v1}}]\!]$ and $[\![\text{Customer}_{\text{v2}}]\!]$, are shown in a tabular format in Figure 4 and Figure 5, respectively.

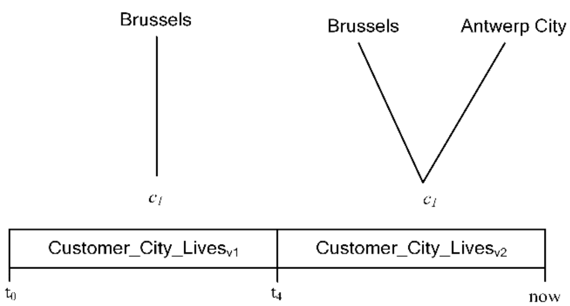
Definition 8 (Aggregation relationship schema). The schema of an aggregation relationship is $R(R_{v_1}, \dots, R_{v_n})$, where R is the aggregation relationship name, and each $R_{v_i}, i = 1, \dots, n,$ is an aggregation relationship version. The schema of each R_{v_i} is $R_{v_i}(L_c, L_p, card, f_{R_i})$, where R_{v_i} is the aggregation relationship version name, L_c, L_p are level versions of levels L and L', respectively, $card \in Card$ specifies the cardinality of the relationship between the child L_c and the parent level L_p, and $f_{R_i}: [\![L_c]\!]_\kappa \to [\![L_p]\!]_\kappa$ is a coercion function as in Def.1. Moreover, each R_{v_i} is defined between the versions of L and L'. Also, if $L = L'$ then $L_c = L_p$, meaning that a recursive aggregation relationship cannot be created between different versions of a level. Finally, the function $levels(R_{v_i})$ returns the tuple (L_c, L_p). ■

Example 3. The schema of aggregation relationship Customer_City_Lives after t_4 is Customer_City_Lives(Customer_City_Lives$_{\text{v1}}$, Customer_City_Lives$_{\text{v2}}$). The schema of each aggregation relationship version is Customer_City_Lives$_{\text{v1}}$ (Customer$_{\text{v1}}$, City$_{\text{v1}}$, m-1, f_{R1}), and Customer_City_Lives$_{\text{v2}}$ (Customer$_{\text{v2}}$, City$_{\text{v1}}$, m-m, f_{R2}), respectively. The aggregation relationship is created between versions of Customer and City active at t_4, which are, Customer$_{\text{v2}}$ and City$_{\text{v1}}$.

Definition 9 (Aggregation relationship instance). The instance $[\![R]\!]$ of an aggregation relationship from Def. 8 is $[\![R]\!]([\![R_{v_1}]\!], \dots, [\![R_{v_n}]\!])$, where $[\![R_{v_i}]\!], i = 1, \dots, n,$ is an instance of each of its versions R_{v_i}. An aggregation relationship instance $[\![R_{v_i}]\!]$ of schema $R_{v_i}(L_c, L_p, card, f_{R_i})$, is a relation defined by $R_{v_i} = [\![\widehat{R_{v_i}}]\!] \cup (\overline{l_c}.k, \overline{l_p}.k)$, where $[\![\widehat{R_{v_i}}]\!] \subset [\![L_c]\!]_\kappa \times [\![L_p]\!]_\kappa, and\ \overline{l_c} \in [\![L_c]\!]$ and $\overline{l_p} \in [\![L_p]\!]$ are the default members of level versions L_c and L_p, respectively. If $l_c \in [\![L_c]\!]_\kappa, l_p \in [\![L_p]\!]_\kappa, and (l_c, l_p) \notin [\![R_{v_i}]\!]$, then $(l_c, f_{R_i}(l_c)) \in [\![R_{v_i}]\!]$, where $f_{R_i}(l_c)$ provides a default parent for the orphan child member l_c. Further, R_{v_i} satisfies the cardinality constraint $card$. Moreover, $[\![R]\!]$ satisfies the aggregation consistency constraint, that is, given versions R_i and R_j of R, $\forall (l_c, l_p) \in [\![R_i]\!], \exists (l_c, l_p') \in [\![R_j]\!]$ and $(l_c, l_p') \in [\![R_i]\!]$.■

Example 4. An aggregation relationship instance for ⟦Customer_City_Lives⟧ consists of the instances of its two versions as follows. $[\![\text{Customer\_City\_Lives}_{\text{v1}}]\!]$ = {(c_1, Brussels),(c_2, Antwerp City)...,(c_n, Brussels), ($c_{default}$, Default)}, and $[\![\text{Customer\_City\_Lives}_{\text{v2}}]\!]$ = {(c_1 ,Brussels), (c_1 , Antwerp),(c_2, Antwerp City)...,(c_n, Brussels), ($c_{default}$, Default)}. The second version of the aggregation relationship allows m-m relationships. As shown in Figure 10, the aggregation consistency constraint ensures that in each version instance, there is at least one common customer to city assignment.

Figure 10 An example instance of two version of Customer_City_Lives.



Definition 10 (Roll-up hierarchy schema). The schema of a roll-up hierarchy is $H(H_{v_1}, \dots, H_{v_n})$, where H is the roll-up hierarchy name, and each $H_{v_i}, i = 1, \dots, n,$ is a hierarchy version. The schema of each roll-up hierarchy version H_{v_i} is $H_{v_i}(R'_1, \dots, R'_m),$ where each $R'_j, j = 1, \dots, m,$ is an aggregation relationship version as in Def. 8, and $R'_j.L_p = R'_{j+1}.L_c, j = 1, \dots, m-1.$ This constraint ensures that the parent level of aggregation relationship version R'_j is the same as the child level of the next relationship version R'_{j+1}, except for the last level of the hierarchy. The *base level* of the hierarchy version H_{v_j} is denoted $L_b = R'_1.L_c$. Also, $L_t = R'_n.L_p$ is called the *top level* of H_{v_j}. Furthermore, given the relation $P = \{level(R'_1), \dots, level(R'_n),\}, P^*_{v_i}$ is the transitive closure of P, meaning that if relations (L_c, L_p) and (L_p, L'_p) exist in P then $(L_c, L'_p) \in P^*_{v_i}$. ■

Example 5. The schema of the roll-up hierarchy Categories after t_4 is Categories(Categories$_{v1}$, Categories$_{v2}$), and the schemas of hierarchy versions are Categories$_{v1}$ (Product_Category$_{v1}$), and Categories$_{v2}$ (Product_SubCategory$_{v1}$, Product_SubCategory$_{v1}$, SubCategory_Category$_{v1}$), respectively.

Definition 11 (Roll-up hierarchy instance). The instance $[\![H]\!]$ of a roll-up hierarchy from Def. 10 is $([\![H_{v_1}]\!], \dots, [\![H_{v_n}]\!])$, where each $[\![H_{v_i}]\!], i = 1, \dots, n,$ is an instance of hierarchy version H_{v_i}. The instance of $[\![H_{v_i}]\!]$ is given by $[\![H_{v_i}]\!] = [\![R'_1]\!] \cup \dots \cup [\![R'_m]\!]$, where $[\![R'_j]\!], j = 1, \dots, m$ is an aggregation relationship version as in Def. 9. Moreover, $[\![H_{v_i}]\!]^*$ denotes the transitive closure of $[\![H_{v_i}]\!]$ meaning that if relations $(l, l'), (l', l'') \in [\![H_{v_i}]\!]$ then relation $(l, l'') \in [\![H_{v_i}]\!]^*$. Further $df_{l_c} = \{l_t \mid l_c \in H_{v_i}.[\![L_b]\!] \wedge l_t \in H_{v_i}.[\![L_t]\!] \wedge (l_c.k, l_t.k) \in [\![H_{v_i}]\!]^*\}$, is the distribution factor of a base level member l_c of hierarchy version H_{v_n}. For all $H_{v_i}, H_{v_j} \in H$, and $\forall(L_c, L_p) \in P^*_{v_i}$ and $(L'_c, L'_p) \in P^*_{v_j}$, where L_c, and L'_c are versions of the level L and L_p, L'_p are versions of level L', the hierarchy consistency constraint holds, that is $\forall(l_c, l_p) \in [\![H_{v_i}]\!]^*, \exists(l'_c, l'_p) \in [\![H_{v_j}]\!]^*$

such that $l_c = l'_c$ and $(l'_c, l'_p) \in [\![H_{v_i}]\!]^*$, where $l_c \in [\![L_c]\!]_\kappa, l'_c \in [\![L'_c]\!]_\kappa, l_p \in [\![L_p]\!]$, and $l'_p \in [\![L'_p]\!]_\kappa$. ■

Example 6. An instance of Categories hierarchy is ($[\![\text{Categories}_{v1}]\!], [\![\text{Categories}_{v2}]\!]$), and $[\![\text{Categories}_{v1}]\!] = [\![\text{Product\_Category}_{v1}]\!]$ and $[\![\text{Categories}_{v2}]\!] = [\![\text{Product\_SubCategory}_{v_1}]\!] \cup [\![\text{SubCategory\_Category}_{v_1}]\!]$. Since it is possible to reach from Product to Category in both hierarchy versions, the hierarchy consistency constraint ensures that both paths get to the same category for a given product.

Definition 12 (Fact version schema). The schema of a fact version F_v is defined by $F_v(K{:}\, b_k, \mathcal{D}_v, \mathcal{M}_v)$, where F is the fact name, K is the key attribute name with a base type $b_k \in \beta$, the tuple $\mathcal{D}_v(D_1, \dots, D_m)$ defines the dimensions of F_v, and each $D_i \in \mathcal{D}, i = 1, \dots, m$, is a pair $D_i(L_i, card_i)$ such that D_i is a dimension name, L_i is a version level name, $card_i \in Card \setminus \{0m - 0m, 0m - m, m - 0m, m - m\}$ specifies the cardinality of the relationship between fact version F_v and level version L_i. The tuple $\mathcal{M}_v(M_1{:}\, O_1, \dots, M_n{:}\, O_n)$ defines the measures of F_v, where each $M_j \in \mathcal{M}, j = 1, \dots, n$, is a measure name that has a base type $b_i \in \beta$. The key attribute name K, a dimension name D, and a measure name M are unique in F_v. The functions $dimNames{:}\, F_v \to \{D_1, \dots, D_m\}$, and $measureNames{:}\, F_v \to \{M_1, \dots, M_n\}$ give the dimensions and measure names of F_v, respectively. ■

Definition 13 (Fact version instance). A fact version instance $[\![F_v]\!]$ of a fact version F_v from Def. 12 is a set of fact members defined by $[\![F_v]\!] \subset \{k \times l_1 \times \dots \times l_m \times m_1 \times \dots \times m_n \mid k \in dom(b_k), l_i \in [\![L_i]\!]_\kappa, m_j \in dom(b_j) \wedge i = 1, \dots, m \wedge j = 1, \dots, n\}$. The tuple $e(f.k, f.l_1, \dots, f.l_m), f \in [\![F_v]\!]$ is an *m-dimensional* cell, which is a placeholder for the values of *n* measures. Each $e.l_i, i = 1, \dots, m$, is called a coordinate value for the dimension D_i of the cell. The fact version instance $[\![F_v]\!]$ satisfies the cardinality constraint like defined above for the aggregation relationship version instances. The set of the key values of a fact instance is denoted by $[\![F_v]\!]_\kappa = \{f.k \mid f \in [\![F_v]\!]\}$. ■

Definition 14 (Metafact schema). The schema of a metafact F_{meta} is $F_{meta}(K{:}\, b_k, \mathcal{D}, \mathcal{M})$, where F_{meta} is a metafact name, K is the name of the key attribute, and it has a base type $b_k \in \beta$, the tuple $\mathcal{D}(D_1, \dots, D_m)$ defines the dimensions of F_{meta}, and each $D_i \in \mathcal{D}, i = 1, \dots, m, is\ a\ tuple\ D_i(L_i, f_{D_i}, card_i)$ such that D_i is a dimension name, L_i is a metalevel name, f_{D_i} is a coercion function, and $card_i \in Card \setminus \{0m - 0m, 0m - m, m - 0m, m - m\}$ specifies the cardinality of the relationship between fact version F_{meta} and level L_i. The tuple $\mathcal{M}\left((M_1{:}\, O_1, f_{M_1}) \dots, (M_n{:}\, O_n, f_{M_m})\right)$ defines the measures of F_{meta}, where each $M_j \in \mathcal{M}, j = 1, \dots, n$, is a measure name that has a base type $b_i \in \beta$, and f_{M_j} is a coercion function. The key name K, a dimension name D, and a measure name M are unique in F_M. The functions $dimNames{:}\, F_{meta} \to \{D_1, \dots, D_m\}$ and $measureNames{:}\, F_{meta} \to \{M_1, \dots, M_n\}$ give the dimensions and measure names of F_{meta}, respectively.

Definition 15 (Metafact instance). The instance $[\![F_{meta}]\!]$ of a metalevel from Def. 14 is a set of fact members defined by $[\![F_{meta}]\!] \subset \{k \times g(l_1) \times ... \times g(l_m) \times h(m_1) \times ... \times h(m_n) \mid k \in dom(b_k) \wedge l_i \in [\![L_i]\!]_\kappa \wedge m_j \in dom(b_j) \wedge i = 1, ..., m \wedge j = 1, ..., n\}$, such that

$$g(l_i) = \begin{cases} f_{D_i}(x_1, ..., x_p), & if\ l_i = \perp \\ l_i, & otherwise \end{cases}$$

and

$$h(m_j) = \begin{cases} f_{M_j}(x_1, ..., x_q), & if\ m_j = \perp \\ m_j, & otherwise \end{cases}$$

Further, $[\![F_{meta}]\!]$ satisfies the key constraint and the cardinality constraints like as defined above for the level version instances and the aggregation relationship instances. The set of the key values of $[\![F_{meta}]\!]$ is denoted by $[\![F_{meta}]\!]_\kappa = \{f.k \mid f \in [\![F_{meta}]\!]\}$.

Definition 16 (Fact schema). The schema of a fact is $F(F_{meta}, \mathcal{F}_v, \chi_D, \chi_M)$, where F is the fact name, F_{meta} is the metafact as defined in Def. 14, $\mathcal{F}_v = \{F_{v_1}, ..., F_{v_n}\}$ is a set of fact versions, and each fact version is as defined in Def. 12, $\chi_D = \{\delta_1, ..., \delta_n\}$ is a set of functions, where a $\delta_i: dimNames(F_{v_i}) \rightarrow dimNames(F_{meta}), i = 1, ..., n$, is a total function. Finally, $\chi_M = \{\mu_1, ..., \mu_n\}$ is a set of functions, where a $\mu_j: measureNames\left(F_{v_j}\right) \rightarrow measureNames(F_{meta}), j = 1, ..., n$, is a total function. Moreover, all fact versions in $\mathcal{F}_v$ have the same key attribute K.

Example 7. At t_4, the schema of fact Sales is $\text{Sales}(\text{Sales}_{\text{meta}}, \mathcal{F}_v = \{\text{Sales}_{\text{v}_1}, \text{Sales}_{\text{v}_2}\}, \chi_D = \{\delta_1, \delta_2\}, \chi_M = \{\mu_1, \mu_2\})$. The schema of $\text{Sales}_{\text{meta}}$ is $\text{Sales}_{\text{meta}}(K, \mathcal{D}, M)$, and K = SalesID :Integer. The dimensions in $\text{Sales}_{\text{meta}}$ are $\mathcal{D}$(Customer, SalesMonth, SalesDate, ShippedDate, Product, Store), and the measures are $\mathcal{M}$(Quantity: Integer, Discount : Decimal, SalesAmount: Integer, NetAmount: Decimal, Freight: Decimal). Similarly, the schema of fact version $\text{Sales}_{\text{v}_2}$ is K = SalesID:Integer, $\mathcal{D}$(Product, Customer, SalesDate, ShipDate), and the measures are $\mathcal{M}$(Quantity: Integer, Discount: Decimal, SalesAmount: Integer, NetAmount: Decimal, Freight: Decimal). Functions $\delta_2 \in \chi_D$ and $\lambda_2 \in \chi_M$ establish the mapping between the dimensions and measures of fact version $\text{Sales}_{\text{v}_2}$ and $\text{Sales}_{\text{meta}}$, respectively.

Definition 17 (Fact instance). The instance $[\![F]\!]$ of a fact F from Def. 16 consists of the instance of its metafact $[\![F_{meta}]\!]$, and the instance $[\![F_v]\!]$ of each of its fact versions $F_v \in \mathcal{F}_v$. The instance $[\![F_{meta}]\!]$ and $[\![F_v]\!]$ are as defined in Def. 15 and 13, respectively. Additionally, each fact member in $[\![F_{meta}]\!]$ is present in $[\![F_v]\!]$, that is, $[\![F_{meta}]\!]_\kappa = [\![F_v]\!]_\kappa$. Furthermore, $\forall f \in [\![F_v]\!], \exists f' \in [\![F_{meta}]\!]$, such that $f.D = f'.D'$, and $f.M = f.M'$, where $(D, D') \in \delta_v$ and $(M, M') \in \mu_v$. ■

Example 8. The fact instance [[Sales]] consists of the instance of meatfact $[\![\text{Sales}_{\text{meta}}]\!]$ (Figure 11) and its version instances $[\![\text{Sales}_{\text{v1}}]\!]$ (Figure 12) and $[\![\text{Sales}_{\text{v2}}]\!]$ (Figure 13). In figures, the fact members loaded after t_4 are shaded in gray.

Figure 11 Instance of metafact $\text{Sales}_{\text{meta}}$ *after* t_4.

SalesKey	Customer	Product	SalesMonth	SalesDate	ShipDate	Store	Quantity	…	Freight
1	c_1	p_1	m_1	t_1	t_2	st_1	10	…	0.0
…	…	…	…	…	…	…	…	…	…
10	c_1	p_3	m_1	t_1	t_1	$st_{default}$	5	…	5.0
…	…	…	…	…	…	…	…	…	…

Figure 12 Instance of fact version Sales_{v1} *after* t_4.

SalesKey	Customer	Product	SalesMonth	Store	Quantity	SalesAmount	NetAmount
1	c_1	p_1	m_1	st_1	10	50	60
…	…	…	…	…	…	…	…
10	c_1	p_3	m_1	$st_{default}$	5	25	27.5
…	…	…	…	…	…	…	…

Figure 13 Instance of fact version Sales_{v2} *after* t_4.

SalesKey	Customer	Product	SalesDate	ShipDate	Quantity	SalesAmount	NetAmount	Freight
1	c_1	p_1	t_1	t_2	10	50	60	0
…	…	…	…	…	…	…	…	…
10	c_1	p_3	t_1	t_1	5	25	27.5	5
…	…	…	…	…	…	…	…	…

Definition 18 (Data warehouse version schema). A DW version schema is $V(\mathcal{L}_v, \mathcal{R}_v, \mathcal{H}_v, \mathcal{F}_v)$, where V is the schema version name, $\mathcal{L}_v = \{L_1, \dots, L_m\}$ is a set of level versions and all level versions in $\mathcal{L}_v$ have a unique name and have the same base type b_k for key attribute A_k. $\mathcal{R}_v = \{R_1, \dots, R_n\}$ is a set of aggregation relationships versions between level versions in $\mathcal{L}_v$, and all aggregation relationship version names are unique in $\mathcal{R}_v$. $\mathcal{H}_v = \{H_1, \dots, H_p\}$ is a set of hierarchies defined over aggregation relationship versions in $\mathcal{R}_v$ and all hierarchy names in $\mathcal{H}_v$ are unique, and $\mathcal{F}_v = \{F_1, \dots, F_q\}$ is a set of fact versions, and all fact version names are unique in $\mathcal{F}_v$. Furthermore, only one version of a level, aggregation relationship, roll-up hierarchy, and fact can be present in $\mathcal{L}_v, \mathcal{R}_v, \mathcal{H}_v$, and $\mathcal{F}_v$, respectively. ■

Example 9. The schema of a DW version V_2 is $V_2(\mathcal{L}_v, \mathcal{R}_v.\mathcal{H}_v, \mathcal{F}_v)$, where $\mathcal{L}_v$ = {Customer_{v2}, City_{v1}, Day_{v1}, Product_{v1}, State_{v1}, Country_{v1}}, $\mathcal{R}_v$ = {$\text{Customer\_City\_Works}_{v1}$, $\text{Customer\_City\_Lives}_{v2}$, $\text{Product\_Category}_{v1}$, City_State_{v1}, $\text{State\_Country}_{v1}$}, $\mathcal{H}_v$ ={Categories_{v2}, Lives_{v_2}, Works_{v1}} and $\mathcal{F}_v$ = {Sales_{v2}}.

Definition 19 (Data warehouse version instance). The instance $[\![V]\!]$ of a DW version V from Def. 18 is $([\![\mathcal{L}_v]\!], [\![\mathcal{R}_v]\!], [\![\mathcal{H}_v]\!], [\![\mathcal{F}_v]\!])$, where $[\![\mathcal{L}_v]\!] = \{[\![L_1]\!], \dots, [\![L_m]\!]\}$ is a set of level version instances, $[\![\mathcal{R}_v]\!] = \{[\![R_1]\!], \dots, [\![R_n]\!]\}$ is a set of aggregation relationship version instances, $[\![\mathcal{H}_v]\!] = \{[\![H_1]\!], \dots, [\![H_p]\!]\}$ is a set of hierarchy version instances, and $[\![\mathcal{F}]\!] = \{[\![F_1]\!], \dots, [\![F_q]\!]\}$ is a set of fact version instances. ■

Definition 20 (Multiversion data warehouse schema). A MVDW schema is $\mathcal{S}(\mathcal{L}, \mathcal{R}, \mathcal{F}, \mathcal{V})$, where

$\mathcal{S}$ is the MVDW schema name, $\mathcal{L} = \{L_1, \dots, L_m\}$ is a set of levels and each $L_i, i = 1, \dots, m$, is a level as in Def. 6. $\mathcal{R} = \{R_1, \dots, R_n\}$ is a set of aggregation relationships and each $R_i, i = 1, \dots, n$, is an aggregation relationship as in Def. 8, $\mathcal{H} = \{H_1, \dots, H_p\}$ is a set of hierarchies and each $H_i, i = 1, \dots, p$, is a hierarchy as defined in Def. 10, $\mathcal{F} = \{F_1, \dots, F_q\}$ is a set of MV facts and each

$F_i, i = 1, \ldots, q$, is a fact as defined in Def. 16, and $\mathcal{V} = \{V_1, \ldots, V_s\}$ is a set of DW versions and each $V_i, i = 1, \ldots, s$, is a DW version as in Def. 18. ■

Definition 20 (Multiversion data warehouse instance). A MVDW instance $\mathcal{I}$ is $(\llbracket\mathcal{L}\rrbracket, \llbracket\mathcal{R}\rrbracket, \llbracket\mathcal{F}\rrbracket, \llbracket\mathcal{V}\rrbracket)$, where $\llbracket\mathcal{L}\rrbracket = \{\llbracket L_1\rrbracket, \ldots, \llbracket L_n\rrbracket\}$ is a set of level instances and each $\llbracket L_i\rrbracket, i = 1, \ldots, m$, is a level instance as defined in Def. 7; $\llbracket\mathcal{R}\rrbracket = \{\llbracket R_1\rrbracket, \ldots, \llbracket R_n\rrbracket\}$ is a set of MV aggregation relationship instances and each $\llbracket R_i\rrbracket, i = 1, \ldots, n$, is an aggregation relationship instance as defined in Def. 9; $\llbracket\mathcal{H}\rrbracket = \{\llbracket H_1\rrbracket, \ldots, \llbracket H_p\rrbracket\}$ is a set of hierarchy instances and each $\llbracket H_k\rrbracket, k = 1, \ldots, p$, is a hierarchy instance as defined in Def.11; $\llbracket\mathcal{F}\rrbracket = \{\llbracket F_1\rrbracket, \ldots, \llbracket F_q\rrbracket\}$ is a set of fact instances and each $\llbracket F_i\rrbracket, i = 1, \ldots, q$, is a fact instance as defined in Def. 17; and $\llbracket\mathcal{V}\rrbracket = \{\llbracket V_1\rrbracket, \ldots, \llbracket V_s\rrbracket\}$ is a set of DW version instances and each $\llbracket V_i\rrbracket, i = 1, \ldots, s$, is a DW version instance as in Def. 19. ■

OLAP Operations in a Multiversion Data Warehouse

Typically, the MD data is exploited using the so-called OLAP operators. The syntax and semantics of these operators are given in (Ahmed, Zimányi, Vaisman, & Wrembel, 2020). Each OLAP operation takes as an input a MD data structure, adds a new fact schema and fact instance to the input MD data structure, and returns the result as a new MD data structure. In this way, the output of one operator can be the input of another operator, and the combination of these operators defines a closed OLAP algebra. It is remarked that each DW version V from Def. 18 is an independent MD data structure. Thus, the OLAP operations can be performed on individual DW versions.

Next, it is shown how the OLAP operators can be applied on a DW version model using the Roll-up operator as an example. The roll-up operation summarizes a measure from a lower level to a higher hierarchy level, using an aggregate function. For example, consider the example MVDW and the query: *Total quantity sold for each product in the city where a store was located*. This query requires a roll-up operation over dimension Store up to level City along hierarchy Location_{v1}. Further, this operation is only available using the DW version V_1 as the level Store was removed in version V_2. The syntax of such an operation is as follows.

- Roll-up:(V_1, Sales_{v1}, Product, Location_{v1}, (Quantity:sum)) → Sales'_{v1}.

The operation adds a new fact Sales'_{v1} to V_1. In Sales'_{v1}, stores are grouped into their respective cities. Since the fact members stored using V_2 did not have dimension Store, a coercion function links such members to default store member.

As shown in Figure 14, the default store is linked to the Default member of level version City_{v1} in the aggregation relationship version Store_City_{v1}. For each city, the Quantity of all products stored in this city is aggregated using the function sum, where all other dimension values are the same. For Store_City_{v1}, and the initial fact Sales from Figure 15, the result is shown in Figure 16. Note that in Sales'_{v1}, keys from level Store are replaced with keys from level version City_{v1} and all measures except Quantity are removed.

Figure 14 Aggregation relationship Store_City$_{v1}$.

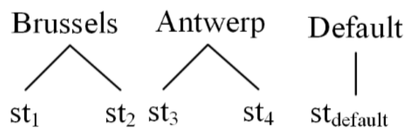


Figure 15 Fact Sales$_{v1}$ *before roll-up.*

SalesKey	Customer	Product	Store	SalesDate	Quantity
1	c_1	p_1	st_1	t_1	5
2	c_1	p_1	st_1	t_1	10
3	c_2	p_2	st_2	t_1	5
4	c_2	p_2	st_3	t_1	8
5	c_1	p_1	$st_{default}$	t_6	5
6	c_1	p_1	$st_{default}$	t_6	6
7	c_2	p_2	$st_{default}$	t_{10}	10

Figure 16 The result of roll-up operation.

SalesKey	Customer	Product	City	SalesDate	Quantity
1	c_1	p_1	Brussels	t_1	15
2	c_1	p_2	Brussels	t_1	5
3	c_2	p_2	Antwerp	t_1	8
4	c_2	p_1	Default	t_6	11
5	c_1	p_2	Default	t_{10}	10

IMPLEMENTING A MULTIVERSION DATA WAREHOUSE IN RELATIONAL DATABASES

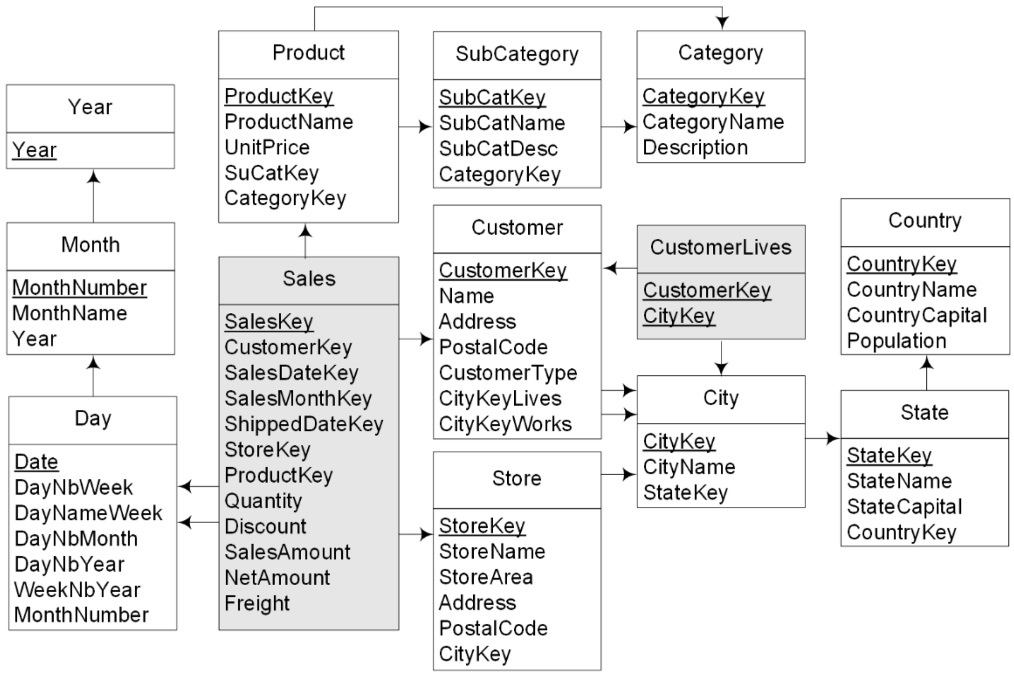
Since most DW are implemented in RDBMS, it appears reasonable to implement a MVDW over a traditional RDBMS. This section shows a translation from the formal MVDW model to a relational schema. Over this model, it is also demonstrated how the OLAP operations can be converted to relational operations implemented in standard SQL.

Relational Schema Mapping

Figure 17 shows the relational representation of metalevels, metafact, and aggregation relationships of the example MVDW after t_4. The translation is explained next.

Level Mapping: For a level $L \in \mathcal{L}$, its metalevel and versions are mapped as follows. The metalevel is mapped to a table T_L that contains all attributes of the level created in all of its versions. A surrogate key is added as the primary key of T_L to map the level's key attribute. Each level version $L_v \in \mathcal{L}_v$ is mapped as a view projecting attributes from T_L that are present in L_v. Additional attributes are added to map the aggregation relationships between level versions, and views corresponding to level versions are also extended with these attributes. For example, in

Figure 17 The metalevels and a metafact in relational schema after schema changes at t_4.



addition to the level attributes, the attributes CityKeyLives and CityKeyWorks are included in level version Customer_{v1} to capture the relationships between Customer_{v1} and City_{v1}.

Fact Mapping: For a fact $F \in \mathcal{F}$, its metafact and versions are mapped as follows. The metafact is mapped to a table T_F that includes as attributes all measures from all of its versions. Further, a surrogate key is added as a primary key to map the key attribute of the fact. Additional attributes are also added to T_F to map the dimensions and link it to all levels providing dimension values. For example, applying the above rule to fact Sales results in a table containing the surrogate keys of all levels it has ever been connected to and the corresponding referential integrity constraints. Each fact version $F_v \in \mathcal{F}_v$ is mapped as a view projecting only the attributes from T_F corresponding to dimensions and measures present in F_v. Figure 17 shows the metalevel and metafact (in gray) in a relational schema after the schema changes at t_4.

Aggregation Relationship Mapping: A relationship between level version of a level with metalevel table T_L and a fact version of a fact with metafact table T_F, or between parent-child level versions with metalevel tables T_p and T_c , respectively, can be mapped in three ways, depending on its cardinality:

- If the relationship is one-to-one, T_F or T_c is extended with all attributes of the T_L or T_p, respectively.
- If the relationship is many-to-one, T_F or T_c is extended with the surrogate key of T_L or T_p, respectively. That is, there is a foreign key in the metafact or the parent metalevel table pointing to the other metastable.

- If the relationship is many-to-many, a new bridge table is created that contains as attribute the surrogate keys of T_p and T_c , respectively. The key of the table is the combination of both surrogate keys. In Figure 17, the bridge table CustomerLives (in gray) captures the aggregation relationship Customer_City_Lives.

The instance of a hierarchy version is not explicitly required to be mapped as it can be obtained by joining the constituting aggregation relationship instances. However, the data dictionary may include the hierarchy schema that can be exploited by an application to generate the SQL to obtain the hierarchy instance automatically. Furthermore, the view definitions ensure that the aggregation consistency constraints are respected.

Coercion Function Mapping: The coercion functions associated with metalevel, metafact, and aggregation relationship versions, are mapped to user-defined functions. These functions are called in an after-insert trigger to obtain the default value if the value of an attribute, dimension, or measure is not available in the inserted row. An SMO may also use a coercion function to obtain the existing rows' missing values after a schema change. For example, after adding attribute CustomerType, the SMO uses the coercion function to get the default customer type for all customers stored using V_1.

Querying a Multiversion Data Warehouse in SQL

This section shows example SQL queries addressing the example MVDW. The example queries below are selected to show how the typical OLAP operations such as roll-up, slice, and dice can be performed in a MVDW.

Query1. *Compute the total sales per category.*

This is a roll-up query that requires aggregating the sales up to level Category_{v1} along the Categories_{v1} hierarchy. By using V_1, the query can be answered as follows.

```
SELECT c.CategoryName , SUM(s.SalesAmount) "Total Amount"
FROM SalesV1 s JOIN ProductV1 p ON s.ProductKey = p.ProductKey
JOIN CategoryV1 c ON p.CategoryKey = c.CategoryKey
GROUP BY c.CategoryName;
```

Since in V_2 the Categories_{v2} hierarchy has an additional level $\text{SubCategories}_{v1}$, the sales per category can be obtained by first performing a roll-up to level SubCategory_{v1} and then to Category_{v1}. These roll-up operations in V_2 are as follows.

```
SELECT c.CategoryName , SUM(s.SalesAmount) "Total Amount"
FROM SalesV2 s JOIN ProductV2 p ON s.ProductKey = p.ProductKey
JOIN SubCategoryV1 sc ON p.SubCategoryKey = sc.SubCategoryKey
JOIN CategoryV1 c ON sc.CategoryKey = c.CategoryKey
GROUP BY c.CategoryName;
```

Query2. *Calculate the total yearly sales per store city, for the beverages category.*

Since level Store was deleted in V_2, this query is only possible using V_1. This query involves dice, project, and roll-up operations. First, fact Sales_{v1} needs to be diced to keep only the sales of the products that belong to the beverage category. Then, all dimensions except Store and SalesDate are removed from the fact. After that, the sales can be aggregated using roll-up operations along Location_{v1} and Calendar_{v1} hierarchies up to levels City_{v1} and Year_{v1}, respectively. Note that the sales that were loaded using V_2 will be aggregated to the default city. The SQL query is given as follows.

```
SELECT ct.CityName , y.Year, SUM( s.SalesAmount) "Total Amount"
FROM SalesV1 s JOIN StoreV1 st ON s.StoreKey = st.StoreKey
JOIN CityV1 ct ON st.CityKey = ct.CityKey
JOIN ProductV1 p ON s.ProductKey = p.ProductKey
JOIN CategoryV1 c ON p.CategoryKey = c.CategoryKey
JOIN MonthV1 m ON s.SalesMonthKey = m.MonthKey
JOIN YearV1 y ON m.YearKey = y.YearKey
AND CategoryName = 'Beverages'
GROUP BY ct.CityName , y.Year ORDER BY ct.CityName , y.Year;
```

Query3. *Calculate the maximum daily sales per subcategory on weekends.*

This query is possible only using V_2 because level SubCategory did not exist in V_1. The query requires first, dicing fact Sales_{v2} to keep only the sales that were made on weekends. Then, all dimensions except Product and SalesDate are removed from the fact. Finally, a roll-up is performed along the Categories_{v2} hierarchy to find the maximum sales per subcategory. Since fact Sales in V_1 stored monthly sales and the coercion function mapped these sales to the first day of each month, they will be considered in this query only if the first day of the month is either Saturday or Sunday.

```
SELECT sc.SubCatName , d.DayNameWeek , MAX(s.SalesAmount) "Total Amount"
FROM SalesV2 s JOIN ProductV2 p ON s.ProductKey = p.ProductKey
JOIN SubCategoryV1 sc ON p.SubCategoryKey = sc.SubCategoryKey
JOIN DayV1 d ON s.SalesDateKey = d.DateKey
AND d.DayNameWeek IN ('Saturday', 'Sunday')
GROUP BY sc.SubCatName , d.DayNameWeek;
```

Query4. *Compute the total yearly sales per customer's city of residence.*

This query can be answered in both versions, and it involves roll-up operations along Live and Calendar hierarchies. Since only one city per customer can be stored in V_1, all sales of a customer will be aggregated to a single city only even though a customer may be living in more than one. The query can be written in SQL for V_1 as below.

```
SELECT c.CityName , y.Year, SUM(s.SalesAmount) "Total Amount"
FROM SalesV1 s JOIN CustomerV1 u ON s.CustomerKey = u.CustomerKey
JOIN CityV1 c ON u.LiveCityKey = c.CityKey
JOIN MonthV1 m ON s.SalesMonthKey = m.MonthKey
JOIN YearV1 y ON y.YearKey = m.YearKey
GROUP BY c.CityName , y.Year ORDER BY c.CityName , y.Year;
```

In V_2, the cardinality of the aggregation relationship between Customer_{v2} and City_{v1} is m-m; therefore, the sales of each customer need to be distributed among the cities of her residence, based on the distribution factor. The SQL query for V_2 is given below.

```
WITH CustCityDF (CustomerKey, df) AS (
SELECT lc.CustomerKey , COUNT(lc.CityKey) "df"
FROM CustomerLiveCityMM lc
GROUP BY lc.CustomerKey )
SELECT c.CityName , y.Year, SUM(s.SalesAmount/cdf.df) "Total Amount"
FROM SalesV2 s JOIN CustomerV2 u ON s.CustomerKey = u.CustomerKey
JOIN CustCityDF cdf ON u.CustomerKey = cdf.CustomerKey
JOIN CustomerLiveCityMM lc ON u.CustomerKey = lc.CustomerKey
JOIN CityV1 c ON lc.CityKey = c.CityKey
JOIN DayV1 d ON s.SalesDateKey = d.DateKey
JOIN MonthV1 m ON d.MonthKey = m.MonthKey
JOIN YearV1 y ON m.YearKey = y.YearKey
GROUP BY c.CityName , y.Year ORDER BY c.CityName , y.Year;
```

RELATED WORK

Managing structural changes is a long-standing issue in database research (Roddick, Craske, & Richards, 1993; Roddick, 1995). Quite a few approaches have been presented to deal with schema changes in relational databases (Curino, Moon, & Zaniolo, 2008; Curino, Moon, Ham, & Zaniolo, 2009; Curino & Zaniolo, 2010; Moon, Curino, Deutsch, Hou, & Zaniolo, 2008; Herrmann, Voigt, Pedersen, & Lehner, 2018), in No-SQL databases (Bonifati, et al., 2019), and in cloud object stores (Armbrust, et al., 2020). Due to space limitations, the remainder of this section reviews the works explicitly dealing with structural changes in DWs.

Schema Evolution in Data Warehouses

The FIESTA (Blaschka, Sapia, & Höfling, 1999; Blaschka, 2001) framework for schema evolution management in MD databases, provides a schema design and maintenance methodology. It includes a high-level evolution algebra to modify the dimensions and fact schema and adapt the existing instance to the evolved schema.

In (Hurtado, Mendelzon, & Vaisman, 1999; Hurtado, Mendelzon, & Vaisman, 1999), the authors present a model to support the updates in level and hierarchy contents and schema, and study the effects of such updates on the materialized views. To manage changes, the model includes a set of content and schema change operators. The content change operators allow adding and deleting new level members and regrouping child members to different parent members. The Generalize, Specialize, Relate, Unrelate, and Delete Level operators are defined for schema changes. Generalize and Specialize operators add a new non-base and base level to a dimension, respectively. Relate and Unrelate operators are used to adding and removing a parallel hierarchy, respectively. The Delete Level operator deletes a given level from the multidimensional schema.

ORE (Jovanovic, Romero, Simitsis, Abelló, & Mayorova, 2014) addresses schema changes due to changes in the analysis requirements. The approach takes two inputs, namely: (a) a domain ontology which represents the concepts and properties of the business model; and (b) the analysis requirements which are called information requirements (IRs). ORE incrementally produces a DW schema that satisfies the IRs.

Kass et al. (Kaas, Pedersen, & Rasmussen, 2004) studied evolution over star and snowflake schemas and how instances change in such cases. The following evolution operations are addressed: insert/delete an attribute into a dimension level, add/remove a level in a dimension, add/remove measure into a fact, and add/remove a dimension into a fact.

Like the values of dimensions members, the definitions of measures may also evolve. In (Goller & Berger, 2015), the authors called such measures as *slowly changing measures* (SCM). Furthermore, they proposed four design solutions to manage the evolution of measure definitions: (a) Conscious do-nothing, (b) Recompute; (c) Proactive versioning; and (d) Lazy amendment. The reader is referred to the bibliography for details.

Schema Versioning in Data Warehouses

TSQL2 (Snodgrass, 1995) has some support for schema versioning, and its schema versioning model is based on the concept of complete schemata, that is, a relation is defined over the union of all the attributes ever defined for it, even including the deleted ones. The database versions can be implemented using the traditional view mechanism on top of the complete schemata of relations. In this sense, the concepts of a metalevel and metafact are similar to that of the complete schemata, and like the table versions, level and fact versions are implemented as views.

In (Grandi, 2002), the author proposes a logical data model to store the multiple database versions in relational databases. Contrary to TSQL2, the data model is based on the idea of a multi-pool storage approach where each version of a relation is stored as a separate physical table. In this way, it is possible to render a single entity with different structural details simultaneously. The information on the schema versions is stored in a catalog consisting of five tables, which combined provide the representation of the schema and associated data to each version. A query language, denotes MSQL, is defined on top of the model, allowing accessing in the same query the schema elements and subsequently the data belonging to multiple schema versions. The language extends SQL to contextualize the names and data references to schema versions.

Golfarelli et al. (Golfarelli, Lechtenbörger, Rizzi, & Vossen, 2006; Rizzi & Golfarelli, 2007) present an approach to managing the content and schema changes and performing so-called cross-version queries. Such queries require data stored in different schema versions. Each DW schema is represented as a directed graph in which nodes represent the dimension attributes or measures, and edges represent the simple, functional dependencies of a canonical cover. The fact node has outgoing edges only. It is connected to all other nodes, representing the dimension attributes and measures. Four graph-based SMOs are used to carry out content and schema changes: add/delete an attribute and add/delete arc.

Furthermore, a so-called augmented schema is used to handle the issue of missing data between consecutive DW versions. When a user creates a new schema version, an augmented schema is also generated: It is the most generic schema containing all the elements from both the new and the old versions. Moreover, in the augmented schema, the user can estimate the value of newly added measures from the existing ones, disaggregate the measure values according to some business rule, manually provide the value of attributes or measures, and check whether the current instances hold for an added relationship. Augmented schema allows the transformation of data between schema versions; therefore, cross-versions queries can be answered.

The model of F. Ravat et al. (Ravat, Teste, & Zurfluh, 2006) consists of a collection of the star schema. Each star schema captures a snapshot of a fact version and multiple dimension versions at their extraction point. The extraction time represents when a dimension member or fact is loaded into the DW and enables the temporally consistent representation of data. It is captured by using a pair of timestamps with dimension members and facts. A set of mapping functions is used to map data from the DW to each star schema.

The COMET (Eder, Koncilia, & Morzy, 2006; Eder, Koncilia, & Kogler, 2002) model associates a period with dimension members, hierarchical relationships, and the schema definitions to maintain the content and schema history. The model allows accurate aggregation (called proportional aggregation) even if an aggregation relationship's cardinality is m-m. It also includes the so-called transformation functions to link the transition between the two following states of dimension members within a single schema version or multiple schema versions. Furthermore, it defines constraints to preserve the data and schema consistency within and across various versions.

In (Wrembel & Bebel, 2005; Wrembel & Morzy, 2006; Wrembel, 2009), the authors propose a MVDW based on the branched versioning model and defined 15 elementary schema changes and 7 instance change operators. A so-called schema derivation transaction is performed to derive the new schema from an existing one. The transaction ensures that the schema derivation step is atomic and creates a consistent and persistent schema. To store the versioning information and to support the cross-version queries, a metamodel is proposed. The proposed MVDW creates a new DW version even in case of content changes. Since the content changes are more frequent than schema changes, this approach may significantly increase the number of versions and the efforts required to maintain the data warehouse. To avoid this problem, the authors suggest grouping multiple changes as a single transaction and execute together to obtain a new version.

The DW evolution framework of Solodovnika et al. (Solodovnikova, 2008; Solodovnikova, Niedrite, & Kozmina, 2015) consists of two modules: the user module, which includes the user reports and queries, and the development module, which consists of metadata and ETL processes. The authors extended the common warehouse metamodel to store the information at the logical and physical levels. The logical metadata holds the information about the versions at a logical level, such as which dimensions, facts etc., are included in a version and its validity. The physical metadata describes the mapping between the MD schema and the relational database objects, i.e., tables, columns etc.

TOLAP-QL (Vaisman, Izquierdo, & Ktenas, 2008) is an SQL-like language that allows metaqueries. A metaquery is posed against the evolving schema instead of the data. For example, in a dimension hierarchy where members of a child level are regrouped to a different parent over time, the query "*how were the members organized at a given time t*" is a metaquery. For query performance, the language has a STORE clause to cache the intermediate result of a query used by subsequent queries. For query optimization, three mechanisms are used, namely join optimization, query pruning, and view materialization. The first one avoids joining the fact with the dimension if the SELECT clause levels are the base level of the dimensions. The second one ignores a fact version if its validity is outside the time implied in the query or the dimension level mentioned in the query did not exist during the lifespan of the version. The third one serves as a cache that stores pre-computed aggregates, thus avoid going back to the original relations and computing the aggregates.

CONCLUSIONS AND FUTURE WORK

Data warehouses (DW) change in their content and schema due to routine business processes, or adoption of new technologies, to name a few. Such changes must be incorporated into a DW for accurate decision-making. Temporal DWs allow managing content changes but cannot deal with the changes in the schema. Multiversion data warehouses (MVDWs) allow managing both content and schema changes; however, their implementation is complicated. This paper extended a multidimensional model (MD) to allow managing schema changes. In this way, the temporal MD model can handle the content changes, and the MV MD model can be used to address the schema changes.

Moreover, to derive various schema versions, the semantics of schema modification operators are given. These operators create the new schema elements and modify the existing data to allow the semantically correct results of OLAP operations. Since each DW version acts as a MD data structure, the traditional OLAP operations can be performed on it. Finally, the translation of the model to a relational representation along with an SQL-based implementation is given.

As a result of changes in the data storage model, the ETL populating the model must also evolve. As future work, the SMOs can be extended to incorporate the ETL evolution. When temporal and MV extension and used together, schema changes may occur, such as making a non-temporal element a temporal one. The semantics of such changes also needs to be defined.

References

Ahmed, W., Zimányi, E., Vaisman, A., & Wrembel, R. (2020). Temporal multidimensional model and OLAP operators. *International Journal of Data Warehousing and Mining, 16*, 51-67.

Armbrust, M., Das, T., Paranjpye, S., Xin, R., Zhu, S., Ghodsi, A., . . . Zaharia, M. (2020). Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *VLDB Endow., 13*, 3411-3424.

Blaschka, M. (2001). FIESTA: A Framework for Schema Evolution in Multidimensional Databases (Abstract). *Datenbank Rundbrief, 27*, 65-66.

Blaschka, M., Sapia, C., & Höfling, G. (1999, 8). On Schema Evolution in Multidimensional Databases. *Int. Conf. on Data Warehousing and Knowledge Discovery, DaWaK* (pp. 153-164). Florence: Springer.

Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., & Voigt, H. (2019). Schema validation and evolution for graph databases. *Int. Conf. on Conceptual Modeling*, (pp. 448-456).

Curino, C. A., & Zaniolo, C. (2010). Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM. *Proc. of the VLDB Endowment, 4*, 117-128.

Curino, C. A., Moon, H. J., & Zaniolo, C. (2008). Graceful database schema evolution: the PRISM workbench. *Proc. of the VLDB Endowment, 1*, 761-772.

Curino, C. A., Moon, H. J., Ham, M., & Zaniolo, C. (2009). The PRISM Workwench: Database Schema Evolution without Tears. *Int. Conf. on Data Engineering, ICDE* (pp. 1523-1526). Shanghai: IEEE.

Eder, J., Koncilia, C., & Kogler, H. (2002). Temporal Data Warehousing: Business Cases and Solutions. *Int. Conf. on Enterprise Information Systems, ICEIS, 1*, pp. 81-88. Ciudad.

Eder, J., Koncilia, C., & Morzy, T. (2006). The COMET Metamodel for Temporal Data Warehouses. *Int. Conf. on Advanced Information Systems Engineering, CAiSE. 2348*, pp. 83-99. Toronto: Springer.

Golfarelli, M., & Rizzi, S. (2009). A survey on temporal data warehousing. *International Journal of Data Warehousing, 5*.

Golfarelli, M., Lechtenbörger, J., Rizzi, S., & Vossen, G. (2006). Schema versioning in data warehouses: Enabling cross-version querying via schema augmentation. *Data & Knowledge Engineering, 59*, 435-459.

Goller, M., & Berger, S. (2015). Handling measurement function changes with Slowly Changing Measures. *Information Systems, 53*, 107-123.

Grandi, F. (2002). A Relational Multi-Schema Data Model and Query Language for Full Support of Schema Versioning. *Decimo Convegno Nazionale su Sistemi Evoluti per Basi di Dati, SEBD*, (pp. 323-336). Portoferraio, Isola d'Elba.

Herrmann, K., Voigt, H., Pedersen, T. B., & Lehner, W. (2018). Multi-schema-version data management: data independence in the twenty-first century. *The VLDB Journal, 27*, 547-571.

Hurtado, C. A., Mendelzon, A. O., & Vaisman, A. A. (1999). Maintaining Data Cubes under Dimension Updates. *Int. Conf. on Data Engineering, ICDE* (pp. 346-355). Sydney: IEEE.

Hurtado, C. A., Mendelzon, A. O., & Vaisman, A. A. (1999). Updating OLAP dimensions. *Int. Workshop on Data Warehousing and OLAP, DOLAP* (pp. 60-66). Kansas City: ACM.

Jovanovic, P., Romero, O., Simitsis, A., Abelló, A., & Mayorova, D. (2014). A requirement-driven approach to the design and evolution of data warehouses. *Information Systems, 44,* 94-119.

Kaas, C., Pedersen, T. B., & Rasmussen, B. (2004). Schema evolution for stars and snowflakes. *Int. Conf. on Enterprise Information Systems, ICEIS,* (pp. 425-433). Porto.

Kimball, R., & Ross, M. (2013). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd ed.). John Wiley & Sons.

Malinowski, E., & Zimányi, E. (2008). A conceptual model for temporal data warehouses and its transformation to the ER and the object-relational models. *Data & Knowledge Engineering, 64,* 101-133.

Merlo, I., Bertino, E., Ferrari, E., & Guerrini, G. (1999). A temporal object-oriented data model with multiple granularities. *Int. Workshop on Temporal Representation and Reasoning. TIME-99,* (pp. 73-81).

Moon, H. J., Curino, C. A., Deutsch, A., Hou, C.-Y., & Zaniolo, C. (2008). Managing and querying transaction-time databases under schema evolution. *Proc. of the VLDB Endowment, 1,* 882-895.

Nargesian, F., Zhu, E., J. Miller, R., Pu, K., & Patricia C. , A. (2019). Data Lake Management: Challenges and Opportunities. *VLDB Endow.,* 1986-1989.

Qiu, D., Li, B., & Su, Z. (2013). An empirical analysis of the co-evolution of schema and code in database applications. *Proc. of the Joint Meeting on Foundations of Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE),* (pp. 125-135).

Ravat, F., Teste, O., & Zurfluh, G. (2006). A multiversion-based multidimensional model. *Int. Conf. on Data Warehousing and Knowledge Discovery, DaWaK* (pp. 65-74). Krakow: Springer.

Rizzi, S., & Golfarelli, M. (2007). X-time: Schema versioning and cross-version querying in data warehouses. *Int. Conf. on Data Engineering, ICDE* (pp. 1471-1472). Istanbul: IEEE.

Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information & Software Technology, 37,* 383-393.

Roddick, J. F., Craske, N. G., & Richards, T. J. (1993). A taxonomy for schema versioning based on the relational and entity relationship models. *Int. Conf. on Entity-Relationship Approach* (pp. 137-148). Springer.

Rundensteiner, E. A., Koeller, A., & Zhang, X. (2000). Maintaining data warehouses over changing information sources. *Communications of the ACM, 43,* 57-62.

Sjøberg, D. (1993). Quantifying schema evolution. *Information and Software Technology, 35,* 35-44.

Snodgrass, R. T. (Ed.). (1995). *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers.

Solodovnikova, D. (2008, 8). Metadata to Support Data Warehouse Evolution. *Int. Conf. on Information Systems Development, ISD* (pp. 627-635). Paphos: Springer.

Solodovnikova, D., Niedrite, L., & Kozmina, N. (2015). Handling Evolving Data Warehouse Requirements. *Proc. of the ADBIS 2015 Short Papers and Workshops* (pp. 334-345). Poitiers: Springer.

Vaisman, A. A., Izquierdo, A., & Ktenas, M. (2008). A web-based architecture for temporal olap. *Int. Journal of Web Engineering and Technology, 4*, 465-494.

Vaisman, A., & Zimányi, E. (2014). *Data Warehouse Systems: Design and Implementation.* Springer.

Vassiliadis, P., Zarras, A. V., & Skoulis, I. (2017). Gravitating to rigidity: Patterns of schema evolution–and its absence–in the lives of tables. *Information Systems*, 24-46.

Wrembel, R. (2009). A survey of managing the evolution of data warehouses. *International Journal of Data Warehousing and Mining, 5,* 24.

Wrembel, R., & Bebel, B. (2005). Metadata management in a multiversion data warehouse. *Proc. of the On the Move Confederated International Conferences, OTM* (pp. 1347-1364). Agia: Springer.

Wrembel, R., & Morzy, T. (2006). Managing And Querying Versions Of Multiversion Data Warehouse. *Int. Conf. on Extending Database Technology, EDBT* (pp. 1121-1124). Munich: Springer.