# Modeling Data Warehouse Schema Evolution over Extended Hierarchy Semantics

Sandipto Banerjee[1] and Karen C. Davis[2]

[1] MicroStrategy, Inc., 1861 International Drive, McLean, VA USA 22102
sbanerjee@microstrategy.com
[2] Electrical & Computer Engineering Dept., University of Cincinnati,
Cincinnati, OH USA 45221-0030
karen.davis@uc.edu

**Abstract.** Models for conceptual design of data warehouse schemas have been proposed, but few researchers have addressed schema evolution in a formal way and none have presented software tools for enforcing the correctness of multidimensional schema evolution operators. We generalize the core features typically found in data warehouse data models, along with modeling extended hierarchy semantics. The advanced features include multiple hierarchies, non-covering hierarchies, non-onto hierarchies, and non-strict hierarchies. We model the constructs in the Uni-level Description Language (ULD) as well as using a multilevel dictionary definition (MDD) approach. The ULD representation provides a formal foundation to specify transformation rules for the semantics of schema evolution operators. The MDD gives a basis for direct implementation in a relational database system; we define model constraints and then use the constraints to maintain integrity when schema evolution operators are applied. This paper contributes a formalism for representing data warehouse schemas and determining the validity of schema evolution operators applied to a schema. We describe a software tool that allows for visualization of the impact of schema evolution through the use of triggers and stored procedures.

**Keywords:** Data warehouse conceptual modeling, data warehouse schema evolution.

## 1 Introduction

There are a number of conceptual models proposed in the literature for data warehouse design. A few examples include Star [11], Snowflake [11], ME/R [42], Cube [45], StarER [43], DFM [18], and EVER [4]. All of these models represent facts containing measures that are aggregated over dimensions. A dimension consists of levels and a hierarchy defines a relationship between the levels. Thus a multidimensional model consists of facts, dimensions, measures, levels, and hierarchies that typically conform to the following constraints:

1. A many-to-one relationship between a fact and a dimension (a fact instance relates to one dimension instance for each dimension, whereas a dimension instance may relate to many fact instances).

S. Spaccapietra et al. (Eds.): Journal on Data Semantics XIII, LNCS 5530, pp. 72–96, 2009.

2. A many-to-one (roll-up) relationship between the levels of a dimension.
3. Hierarchies in a dimension have a single path to roll-up or drill-down.

We refer to these features as the core features of a data warehouse conceptual model.

Pedersen and Jensen [35, 36], Hümmer et al. [22], and Tsois et al. [44] discuss shortcomings of traditional models for adequately capturing real-world scenarios. Each paper presents a different set of features; we adopt four of the most common to extend the traditional semantics of hierarchies in data warehouses. In our work, we consider the following:

- *multiple hierarchies*: a dimension can have multiple paths to roll-up or drill-down information.
- *non-covering hierarchies*: a parent level in a non-covering hierarchy is an ancestor (not immediate parent) of the child level (our graphical convention is illustrated in Figure 1(a)).
- *non-onto hierarchies*: an instance of a parent level in a dimension can exist without a corresponding data instance in the child level to drill-down to (Figure 1(b)).
- *non-strict hierarchies*: two levels in a dimension can have a many-to-many relationship (Figure 1(c)).

Non-covering, non-onto, and non-strict may be arbitrarily combined; notation for a non-covering and non-strict relationship (the case of a many-to-many relationship between an ancestor and a child level) is shown as one example in Figure 1(d)).

(a) non-covering (b) non-onto (c) non-strict (d) non-covering and non-strict

**Fig. 1.** Notation for Extended Hierarchies in Dimensions

In order to illustrate the extended hierarchy semantics and schema evolution (later in the paper), we combine three examples from the literature. The schema shown in Figure 2 uses a modified DFM notation; the dimension name is added as a first level in order to use a different name for the dimension rather than the default value of the root level. As in DFM, the levels are shown with the finer granularity rolling up to the coarser granularity, e.g., *Co-ordinate* is the finest granularity in the *Location* dimension and *Country* is the most general. Golfarelli et al. present an example of a *Sale* fact schema with measures *Qty Sold, Revenue*, and *No. of Customers* [19]. Hurtado et al. present an example of a *Product* dimension [25]. The *Product* dimension models multiple hierarchies because the level *Corporation* drills-down to *Company* or *Category* and the level *Item* rolls-up to *Brand* or *Category*. The paths of the dimension that describe the roll-up from the lowest to the highest hierarchy level are *Item, Category, Corporation* and *Item, Brand, Company, Corporation.*

Jensen et al. present an example of *Time* and *Location* dimensions [26]. The *Location* hierarchies are defined such that *Country* drills-down to *Province*, and *Province* drills-down to *City*, for example. A non-onto hierarchy is defined for *City* and *IP add* because some cities do not have an internet provider address. Three non-strict hierarchies are defined: (1) between *District* and *Street* because a street might belong to many districts, (2) between *City* and *Cell* because a cell can be shared by many cities, and (3) between *Province* and *Cell* because a cell can be shared by many provinces. Three non-covering hierarchies are defined: (1) between *District* and *Co-ordinate* because some co-ordinates may not be in streets, (2) between *Province* and *District* because some districts may not belong to a city, and (3) between *City* and *Co-ordinate* because some co-ordinates may not be in a street or a district.

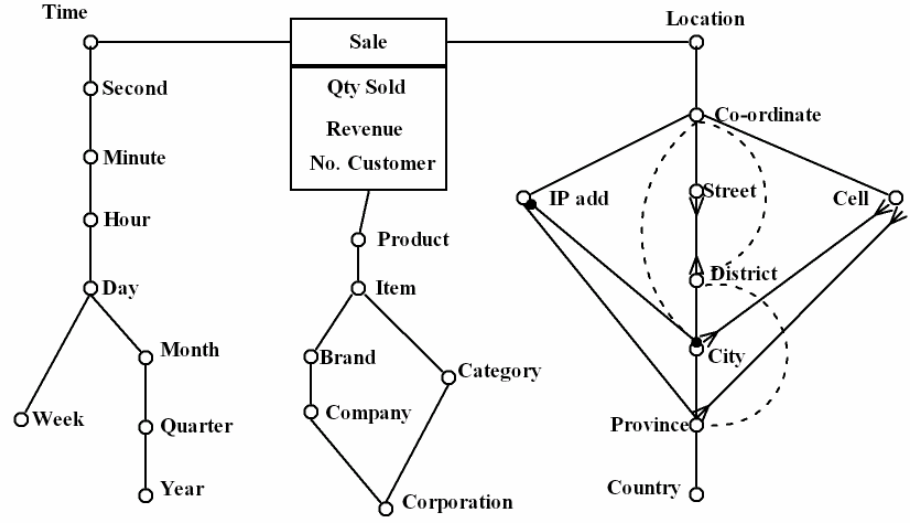


**Fig. 2.** Sale Schema

The diversity of models and schemas in practice motivates use of schema/model management tools such as a multilevel dictionary approach [1] to allow interoperability as well as generic reporting and analysis tools. In this paper, we define a formal metamodel for data warehouse core features using ULD [5, 6, 7] along with a multilevel dictionary definition (MDD). The ULD definition provides formal semantics while the MDD structures can be directly implemented in a relational database system. In addition to the basic structural constructs of a data warehouse model, we also define schema evolution operators and the constraints that must be satisfied to ensure schema correctness. The correctness of these operators is enforced by stored procedures and triggers.

It is important to investigate and specify schema evolution for several reasons. The data warehouse environment is dynamic; a few examples include changes due to changing user needs (e.g., requests for additional information or faster responses) and changing sources. Often data warehouses are initially designed as data marts. Growth, extension, and modification are expected to occur with a data mart.

Previously authors have discussed operators supporting schema evolution over the core data warehouse features. The schema evolution operators defined by Quix [39] and Bouzeghoub and Kedad [8] are primarily concerned with view modification and maintenance and are not further considered here. Our approach focuses on schema correctness rather than synchronization of data with source schemas. Blaschka et al. [9] propose a formal model that is conceptually similar to ours and use an algebra to define schema evolution. Hurtado et al. [25] focus on changes to dimensions only. Chen et al. [10] and Golfarelli et al. [17] give models that rely on general schema evolution operators. Chen et al. have operators for adding, deleting, and renaming tables and attributes, while Golfarelli et al. introduce a graph-based formalism that allows adding and deleting nodes and edges. The nodes represent entities (facts, measures, and dimensions) while the edges represent functional dependencies. None of these data warehouse schema evolution papers introduce a tool for enforcing changes as we do here. The tool allows for rapid creation of a data warehouse schema and immediate checking of the validity of schema evolution operations.

Although authors have discussed operators supporting data warehouse schema evolution [8, 9, 10, 17, 25, 28, 39], none of them consider advanced data warehouse features such as non-strict hierarchies, non-covering hierarchies, and non-onto hierarchies. Extending the semantics of hierarchies increases the expressive power of the model and thus creates new challenges for enforcing correctness when schemas change back and forth between traditional and extended semantics. For example, if a non-covering hierarchy is deleted, such as the relationship between *Province* and *District* (where a district may not belong to a city, but it may belong to a province), then a valid hierarchy must be in place between the affected levels. Since an alternate path is defined such that a district can roll-up to a city, and a city rolls-up to a province, the deletion of the non-covering hierarchy is valid. Additional operators to adjust the data instances would be required but are not the focus of this paper.

In Sections 2 and 3, we introduce a formal model and constraints for extended hierarchy semantics. In Section 4, we describe a multilevel dictionary implementation of the model and use the *Sale* schema in Figure 2 to illustrate schema evolution operations. We summarize our contributions and discuss directions for future work in Section 5.

## 2 Model

The Uni-level Description Language (ULD) [5, 6, 7] provides a metadata description technique that can be used to represent a broad range of data models. ULD has been used to represent a variety of models such as ER, RDF and XML as well as provide the basis for a universal browsing tool for information originating in diverse data models.

We use a ULD as well as a Multilevel Dictionary Definition (MDD) approach [1] to define our model and schema evolution operators. ULD facilitates modeling as it provides a uniform representation of data, schema, model, and metamodel layers, and their interrelationships. The ULD representation also provides a formal foundation to specify transformation rules for the semantics of schema evolution operators. The MDD supports description of target models and schemas in a tabular format. The MDD gives a basis for direct implementation of the constructs of data models/schemas in a relational database system. Triggers and stored procedures can be

used to implement the semantics of schema creation and evolution that we formally define in the ULD. Essentially, we first define the constructs and their semantics formally in ULD, then we represent the constructs in MDD for ease of understanding and implementation, and finally we implement the MDD constructs in a relational database system using triggers and stored procedures to enforce semantics.

The ULD metamodel is used to define the core features of a data warehouse. Figure 3 summarizes the formal definition and highlights the extended features in italics. The ULD construct types we use are $set_{ct}$ and $struct_{ct}$, where the former represents a set of objects and the latter represents a structured object with subcomponents. The notation *"construct c := [x => y]"* represents the construct type $struct_{ct}$, where the expression "*x => y*" represents a component of the construct *c, x* is called the *component selector*, and *y* is the type of the component. The $set_{ct}$ construct type is represented as *set-of*. For example, a *FactSet* is defined as a *set-of Fact*, where a *Fact* is a $struct_{ct}$ with subcomponents for name, measures, attributes, and primary key fields. Constructs can be related to each other by the conformance (::) construct. A conformance construct can be further specified using first-order logic constraints. We use the notation "[*n:1*]" to represent the cardinality between related constructs. For example, the cardinality between levels in a *Hierarchy* is "[*1:n*]" because a parent can have *n* children and a child can have *1* parent. A *Fact* instance relates to one instance of a *Dimension* (*"Dimension cardinality [n:1]")* while a *Dimension* has "[*1:n*]" cardinality to a *Fact*. A non-strict hierarchy (*NShierarchy*) has a many-to-many ("[*m:n*]") *Level cardinality*.

construct Schema := [Sname => uld-string, Sfact =>FactSet, Sdimension => DimensionSet]
construct FactSet := set-of Fact
construct DimensionSet := set-of Dimension
construct Fact := [Fname => uld-string, Fattribute => AttributeSet, Fmeasure => MeasureSet,
FPkey => FactPKeySet] :: Dimension cardinality [n:1]
construct MeasureSet := set-of Measure
construct Measure := [Mname => uld-string, Mdomain => number]
construct FactPKeySet := set-of FactPkey
construct FactPkey := [SubkeyName => uld-string]
construct AttributeSet := set-of Attribute
construct Attribute := [AttributeName => uld-string]
construct LevelSet := set-of Level
construct HierarchySet := set-of Hierarchy
construct Level := [Lname => uld-string]
construct Hierarchy := [Hparent => Level, Hchild => Level]:: Level cardinality [1:n]
construct Cube := [Cname => uld-string, Cfact => Fact, Cdimension => DimensionSet]
construct Dimension := [Dname => uld-string, Dlevel => LevelSet, DPkey => uld-string,
Dhierarchy => HierarchySet, *Dpath => PathSet*]:: Fact cardinality [1:n]
*construct PathSet := set-of Path*
*construct Path := set-of PathHierarchy*
*construct PathHierarchy := [PHierarchy => Hierarchy]*
*construct NChierarchy := [NCparent => Level, NCchild => Level]::PathHierarchy*
*construct NOhierarchy := [NOparent => Level, NOchild => Level]::Hierarchy*
*construct NShierarchy := [NSparent => Level, NSchild => Level]:: Hierarchy :: Level cardinality [n:m]*

**Fig. 3.** Data Warehouse Constructs in ULD

A data warehouse schema consists of facts and dimensions. A *Schema* construct is defined by *Sname*, a unique name, *Sfact*, a set of all facts in a schema, and *Sdimension,* a set of all dimensions in a schema. A fact consists of measures, attributes and a primary key. A *Fact* construct is defined by *Fname*, *Fmeasure*, *Fattribute* and *FPkey*. *Fname* is a unique name, and *Fmeasure* is of the type *MeasureSet* that represents a set of measures in a fact. The *Measure* construct has a unique name and a domain *number*. *Fattribute* is of the type *AttributeSet* that represents a set of attributes in a fact. Attributes are descriptive (non-aggregable) while measures can be aggregated. *FPkey* is of the type *FactPKeySet* that represents a set of subkeys that form the concatenated primary key for a fact.

A dimension in the core model consists of levels, hierarchies and a primary key. A *Dimension* construct is defined by *Dname, Dlevel, DPkey*, and *Dhierarchy* levels. A *Dname* indicates a unique name for a dimension. A *Dlevel* is of the type *LevelSet* that represents a set of levels in a dimension. *DPkey* is the primary key for a dimension. It can be extended to be represented as a concatenated key similar to the primary key of the *Fact* construct. *Dhierarchy* is of the type *HierarchySet* that represents a set of hierarchies among the levels of a dimension.

A *Level* construct consists of a unique name and represents the dimension level over which the measures are aggregated. The roll-up or drill-down relationship between two levels of a dimension is captured by the construct *Hierarchy*. The construct *Hierarchy* consists of *Hparent* and *Hchild*. *Hchild* and *Hparent* convey the information that a child level rolls-up to the parent level.

To represent a dimension with multiple hierarchies the construct definition of a dimension is modified to include *Dpath* that represents a hierarchy path in a dimension. A *Dpath* construct is of the type *PathSet*, a set of paths in a dimension. A unique name, a set of levels, a set of hierarchies, a set of paths over which the data is aggregated, and a primary key define an extended dimension construct. A hierarchy instance consists of an instance of a parent level, instance of child level, and a conformance relationship to a hierarchy. The conformance relationship maps the hierarchy instance to a hierarchy construct.

A *Cube* construct represents a fact and the dimensions connected to a fact. Some authors also define the cube as a view [8, 39]. In our research we define a cube by *Cname*, *Cfact*, and *Cdimension*. *Cname* is a unique name identifying a cube. *Cfact* is a fact and *Cdimension* is a set of dimensions that are connected to that fact. When a cube is created, the measures of the fact are aggregated over the levels of the dimensions. Consider a scenario where a cube called *SalesCube* is created when a *Sales* fact with measures *Cost Price* and *Sales Price* is connected to the dimensions *Time* and *Product*. The levels of the *Time* dimension are *Year, Month* and *Day* and the levels of the *Product* dimension are *Product Type* and *Product Color*. When a cube is created the measures *Cost Price* and *Sales Price* are aggregated over the levels *Year, Month*, *Day, Product Type,* and *Product Color.*

We represent a non-covering hierarchy by mapping the instances of the levels that exhibit non-covering hierarchy behavior [31]. We define a non-covering hierarchy between two levels as:

*construct NChierarchy := [NCparent => Level, NCchild => Level]::Path*

The *NChierarchy* construct represents a non-covering hierarchy between two levels along a path in the dimension. It consists of a parent level, a child level and a conformance relation to a path. For example, non-covering hierarchy is defined for the levels *Country, City* and a conformance relation to the path *p1 (Country, State, City)*. The path represents a set of hierarchies in a dimension. The parent and child level in the non-covering hierarchy have a conformance relationship to a dimension path such that the parent level is an ancestor of the child level and not a direct parent.

The construct *NCInstance* is called a non-covering hierarchy instance and represents the instances of the levels that exhibit non-covering hierarchy semantics. The construct *NCInstance* consists of a parent level instance and a child level instance and a conformance relationship to a non-covering hierarchy. The difference between a non-covering hierarchy instance and a direct hierarchy instance is the conformance relationship.

A non-onto hierarchy occurs when an instance of a parent level does not have an instance of the child level to drill-down to. We define a non-onto hierarchy between two levels as:

*construct NOhierarchy := [NOparent => Level]:: Hierarchy*

The *NOhierarchy* construct represents a non-onto hierarchy and consists of a parent level and a conformance relation to an existing hierarchy in a dimension. The parent level represents the level that has instances that cannot drill-down to instances of the child level in a hierarchy.

A non-strict hierarchy occurs when there is a many-to-many relationship between the parent and child level in a hierarchy. A non-strict hierarchy between two levels of a dimension is defined as:

*construct NShierarchy := [NSparent => Level, NSchild => Level]:: Hierarchy :: Level cardinality [m:n]*

The *NShierarchy* construct represents a non-strict hierarchy and consists of a parent level and child level and a conformance relation to an existing hierarchy in a dimension. The parent level represents the level where an instance drills-down to more than one instance of the child level in a hierarchy.

The correctness and consistency of the constructs in our model are maintained by introducing constraints discussed in the next section.

## 3 Constraints

Addition/deletion constraints, represented in first-order logic, are used to maintain the correctness of the model when a fact, dimension, level, hierarchy or cube is added/deleted. In this section, the core constraints for addition and deletion are introduced first, followed by a discussion of the constraints for the extended hierarchy semantics.

Addition constraints AC1-AC4 in Table 1 enforce the semantics that a fact has to be added to an existing schema, a measure must belong to a fact, a dimension belongs to a schema, and a level can only be added to an existing dimension, respectively, for

the core semantics. As an example, AC1 states that for any *x* that is a construct instance of a fact (*c-inst(x, Fact)*), there must be a schema *z* (*c-inst(z, Schema)*) that has a set of facts as a component (*member-of(Sfact, z)*) to which *x* is an instance (*c-inst(x, Sfact)*) if *x* is to be added as a fact in schema *z*.

**Table 1.** Addition Constraints (Core Features)

| ID | Construct | Constraint |
|---|---|---|
| AC1 | Fact | $\forall(x)\ \exists(z)$ *c-inst(x, Fact)* $\wedge$ *c-inst(z, Schema)* $\wedge$ *member-of(Sfact, z)* $\wedge$ *addFact(x, z)* $\rightarrow$ *c-inst(x, Sfact)* |
| AC2 | Measure | $\forall(x)\ \exists(y)$ *c-inst(x, Measure)* $\wedge$ *c-inst(y, Fact)* $\wedge$ *member-of(Fmeasure, y)* $\wedge$ *addMeasure(x, y)* $\rightarrow$ *c-inst(x, Fmeasure)* |
| AC3 | Dimension | $\forall(x)\ \exists(z)$ *c-inst(x, Dimension)* $\wedge$ *c-inst(z, schema)* $\wedge$ *member-of(SDimension, z)* $\wedge$ *addDimension(x, z)* $\rightarrow$ *c-inst(x, SDimension)* |
| AC4 | Level | $\forall(x)\ \exists(y)$ *c-inst(x, Level)* $\wedge$ *c-inst(y, Dimension)* $\wedge$ *member-of(Dlevel, y)* $\wedge$ *addLevel(x, y)* $\rightarrow$ *c-inst(x, Dlevel)* |
| AC5 (a) | Hierarchy | $\forall(x)\ \exists(y)$ *c-inst(x, Hierarchy)* $\wedge$ *c-inst(y, Dimension)* $\wedge$ *member-of(Dhierarchy, y)* $\wedge$ *addHierarchy(x, y)* $\rightarrow$ *c-inst(x, Dhierarchy)* |
| (b) | | $\forall(x, y)\ \exists(z)$ *c-inst(x, Hierarchy)* $\wedge$ *member-of(Hparent, x)* $\wedge$ *c-inst(y, Hparent)* $\wedge$ *c-inst(z, Dimension)* $\wedge$ *member-of(Dlevel, z)* $\wedge$ *addHierarchy(x, z)* $\rightarrow$ *c-inst(y, Dlevel)* |
| (c) | | $\forall(x, y)\ \exists(z)$ *c-inst(x, Hierarchy)* $\wedge$ *member-of(Hchild, x)* $\wedge$ *c-inst(y, Hchild)* $\wedge$ *c-inst(z, Dimension)* $\wedge$ *member-of(Dlevel, z)* $\wedge$ *addHierarchy(x, z)* $\rightarrow$ *c-inst(y, Dlevel)* |
| (d) | | $\forall(x, a, b, c)$ *c-inst(x, Hierarchy)* $\wedge$ *member-of(Hparent, x)* $\wedge$ *member-of(Hchild, x)* $\wedge$ *c-inst(a, Hparent)* $\wedge$ *c-inst(b, Hparent)* $\wedge$ *c-inst(c, Hchild)* $\wedge$ *isParent(a, c)* $\wedge$ *isParent(b, c)* $\rightarrow$ *equivalent(a, b)* |
| (e) | | $\forall(x, a, b, c)$ *c-inst(x, Hierarchy)* $\wedge$ *member-of(Hparent, x)* $\wedge$ *member-of(Hchild, x)* $\wedge$ *c-inst(a, Hchild)* $\wedge$ *c-inst(b, Hchild)* $\wedge$ *c-inst(c, Hparent)* $\wedge$ *isParent(c, a)* $\wedge$ *isParent(c, b)* $\rightarrow$ *equivalent(a, b)* |
| AC6 (a) | Cube | $\forall(x)\ \exists(z)$ *c-inst(x, Cube)* $\wedge$ *member-of(Cfact, x)* $\wedge$ *addFact(z, x)* $\rightarrow$ *c-inst(z, Fact)* |
| (b) | | $\forall(x)\ \exists(z)$ *c-inst(x, Cube)* $\wedge$ *member-of(Cdimension, x)* $\wedge$ *addDimension(z, x)* $\rightarrow$ *c-inst(z, Dimension)* |
| (c) | | $\forall(x, y)\ \exists(a, b)$ *c-inst(x, Fact)* $\wedge$ *member-of(SubkeyName, x)* $\wedge$ *c-inst(y, SubkeyName)* $\wedge$ *c-inst(a, Dimension)* $\wedge$ *member-of(DPkey, a)* $\wedge$ *c-inst(b, DPkey)* $\wedge$ *ConnectDimensionToFact(a, x)* $\rightarrow$ *c-inst(b, SubkeyName)* |

The addition constraints for the *Hierarchy* construct are:

AC5a. A hierarchy can only be added to an existing dimension.
AC5b. A parent level in a hierarchy is a level in the dimension.
AC5c. A child level in a hierarchy is a level in the dimension.
AC5d. A level in a hierarchy has at most one parent level.
AC5e. A level in a hierarchy has at most one child level.

Constraints AC5(d) and (e) enforce roll-up/drill-down along a single path in a dimension; this constraint is relaxed when we discuss support for multiple hierarchies. The addition constraints for the Cube construct are:

AC6a. To define a valid cube construct, there must be a corresponding fact construct in the schema.
AC6b. A cube dimension can be created if the dimensions connected to the fact exist in the schema.
AC6c. A dimension can be connected to a fact in a cube if the subkey of the concatenated primary key of a fact is a foreign key reference to the primary key of the dimension.

We use familiar relational model terminology in Constraint AC6(c) to simplify the discussion. To be more general, set and functional dependency notation could be used here.

As a concrete example to illustrate an addition constraint, consider a *Time* dimension. If we wish to create a child level called *Month* that rolls-up to *Quarter*, both AC5(b) and AC5(c) would have to be satisfied. We use AC5(c) to illustrate the components of the rule. Let *z* be the *Dimension* with name "Time" and set of levels, *Dlevel*, with values {"Second," "Minute," "Hour," "Day," "Week," "Month," "Quarter," "Year"}. In order to create a hierarchy *x* (a pair of levels) in *z*, where the child component of *x* is *y*, *y* must exist as a level in the schema. Since "Month" is an element of *Dlevel*, creating part of the hierarchy called *x* with child *y* ("Month") is valid for the *Dimension* "Time."

Valid deletions must satisfy the following conditions (DC1-DC4 in Table 2). DC1 allows a level in a dimension to be deleted if it is not a parent nor a child level in a hierarchy. DC2 specifies that a dimension can be deleted from the schema if there are no hierarchies or levels of the dimension, and the dimension is not part of a cube. DC3 states that a fact can be deleted from the schema if there are no corresponding cubes based on that fact. DC4 regards deletion of a cube; a cube can be deleted if there are no fact and dimensions in the cube. Schema evolution operators build on these constraints to delete constructs that depend on other constructs. The deletion constraint for a *Level* requires that levels participating in hierarchies cannot be deleted; hierarchies must be removed first using the schema evolution operator *DeleteHierarchyInDimension* (described in Section 4.2). As an example, deleting a dimension from a schema would first require deleting the hierarchies, then the levels, followed by removing the dimension from any cubes in which it participates, then deleting the dimension by removing it from the schema.

The addition constraints for the extended hierarchy semantics are summarized in Table 3 (deletion constraints are in Table 4) and described below. We show the modified hierarchy constraint (AC5d′ and AC5e′) and AC7 to support multiple hierarchies. Constraints AC8-AC10 support non-covering hierarchies, non-onto hierarchies, and non-strict hierarchies. Previous authors have defined similar constraints in terms of cardinality (e.g., Tryfona et al. [43] and Malinowski and Zimányi [31]); we use predicates as a higher level of abstraction here to define constraints and schema evolution operators.

**Table 2.** Deletion Constraints (Core Features)

| ID | Construct | Constraint |
|---|---|---|
| DC1 (a) | Level | $\forall(x, y)\ \exists(z)$ *c-inst(x, Dimension)* $\wedge$ *member-of(DLevel, x)* $\wedge$ *c-inst(y, DLevel)* $\wedge$ *IsHierarchy(z, x)* $\wedge$ *member-of(ParentLevel, z)* $\wedge \neg$*c-inst(y, ParentLevel)* $\rightarrow$ *delete(y, DLevel)* |
| (b) | | $\forall(x, y)\ \exists(z)$ *c-inst(x, Dimension)* $\wedge$ *member-of(DLevel, x)* $\wedge$ *c-inst(y, DLevel)* $\wedge$ *IsHierarchy(z, x)* $\wedge$ *member-of(ChildLevel, z)* $\wedge \neg$*c-inst(y, ChildLevel)* $\rightarrow$ *delete(y, DLevel)* |
| DC2 (a) | Dimension | $\forall$ *(x, y) c-inst(x, Dimension)* $\wedge$ *member-of(DHierarchy, x)* $\wedge \neg$*c-inst(y, DHierarchy)* $\rightarrow$ *delete(x, Dimension)* |
| (b) | | $\forall$ *(x, y) c-inst(x, Dimension)* $\wedge$ *member-of(DLevel, x)* $\wedge \neg$*c-inst(y, DLevel)* $\rightarrow$ *delete(x, Dimension)* |
| (c) | | $\forall$ *(x, y) c-inst(x, Dimension)* $\wedge$ *c-inst(y, Cube)* $\wedge$ *member-of(CDimension, y)* $\wedge \neg$*c-inst(x, CDimension)* $\rightarrow$ *delete(x, Dimension)* |
| DC3 | Fact | $\forall$ *(x, y) c-inst(x, Fact)* $\wedge$ *c-inst(y, Cube)* $\wedge$ *member-of(Cfact, y)* $\wedge \neg$*c-inst(x, Cfact)* $\rightarrow$ *delete(x, Fact)* |
| DC4 (a) | Cube | $\forall$ *(x, y, z) c-inst(x, Cube)* $\wedge$ *member-of(Cfact, x)* $\wedge$ *c-inst(y, Cfact)* $\wedge$ *member-of(Cdimension, x)* $\wedge$ *c-inst(z, Cdimension)* $\wedge \neg$*c-inst(y, Cfact)* $\wedge \neg$*c-inst(z, Cdimension)* $\rightarrow$ *delete(x, Cube)* |
| (b) | | $\forall$ *(x, z) c-inst(x, Cube)* $\wedge$ *member-of(Cfact, x)* $\wedge$ *member-of(Cdimension, x)* $\wedge \neg$*c-inst(z, Cdimension)* $\rightarrow$ *delete(x, Cube)* |

In a traditional data warehouse model, the levels of a dimension share a direct relationship and the roll-up/drill-down is along a straight path. However, in real world situations, it is possible that a level has more than one child or parent level. Data models have been proposed to support *multiple hierarchies* in a dimension [2, 3, 14, 15, 16, 19, 23, 24, 27, 29, 30, 31, 35, 37, 42, 43, 44]. To represent a dimension with multiple hierarchies, the ULD construct *Dimension* is modified to include *Dpath.* A hierarchy path of a dimension, *Dpath*, represents all possible combinations of roll-up relationships from the bottom-most child level to the top-most parent level.

The addition constraints are modified to support multiple hierarchies in a dimension by ensuring that a parent level in a hierarchy has at least one child level instead of exactly one (AC5(d′)), and a child level has at least one parent (AC5(e′)). We introduce addition and deletion constraints for the path construct (AC7 and DC5, respectively.) The hierarchies of a path in a dimension must be hierarchies of the dimension. A path of a dimension is modified when a hierarchy is added, deleted or modified.

As an example, consider AC10(a) for adding a non-strict hierarchy. There must be a dimension hierarchy in the schema that the *NShierarchy* conforms to; the *NShierarchy* is created from an existing one-to-many hierarchy (parent, child) that becomes many-to-many. This is specified by the constraint and enforced by the schema evolution operator *AddNonStrictHierarchy* (described in Section 4.2).

A *non-covering hierarchy* or ragged hierarchy occurs when an instance of a level rolls-up to a level that is not a direct parent of that level [26, 31, 36]. We model the

**Table 3.** Addition Constraints (Extended Hierarchy)

| ID | Construct | Constraint |
|---|---|---|
| AC5 (d′) | Hierarchy | *∀ (x, a) ∃(b) c-inst(x, Hierarchy) /\ member-of(Hparent, x) /\ c-inst(a, Hparent) → c-inst(b, Hchild)* |
| (e′) | | *∀ (x, a) ∃(b) c-inst(x, Hierarchy) /\ member-of(Hchild, x) /\ c-inst(a, Hchild) → c-inst(b, Hparent)* |
| AC7 | Path | *∀(a,b,c,d) ∃(e) c-inst(a, Dimension) /\ member-of(Dpath, a) /\ c-inst(b, Dpath) /\ member-of(PathHierarchy, b) /\ c-inst(c, PathHierarchy) /\ member-of(Dhierarchy, a) /\ c-inst(e, Dhierarchy) → equivalent(c, e)* |
| AC8 (a) | Non-covering hierarchy | *∀(x, y) ∃(z) c-inst(x, NChierarchy) /\ c-inst(y, Dimension) /\ member-of(Dhierarchy, y) /\ c-inst(z, Dhierarchy) → conformance(x, z)* |
| (b) | | *∀(x, y) ∃(z) c-inst(x, NChierarchy) /\ member-of(NCparent, x) /\ c-inst(y, path) /\ conformance(x, y) /\ hierarchy(z, y) /\ member-of(Hparent, z) /\ c-inst(x, Hparent) → non-coveringparent(y, x)* |
| (c) | | *∀(x, y) ∃(z) c-inst(x, NChierarchy) /\ member-of(NCchild, x) /\ c-inst(y, path) /\ conformance(x, y) /\ hierarchy(z, y) /\ member-of(Hchild, z) /\ c-inst(x, Hchild) → non-coveringchild(y, x)* |
| (d) | | *∀(a,b,c, x) non-coveringparent(b, a) /\ non-coveringchild(c, a) /\ c-inst(x, Hierarchy) /\ member-of(Hparent, x) /\ member-of(Hchild, x) /\ ¬c-inst(b, Hparent) /\ ¬c-inst(c, Hchild) → c-inst(a, NChierarchy)* |
| AC9 (a) | Non-onto hierarchy | *∀(x, y) ∃(z) c-inst(x, NOhierarchy) /\ c-inst(y, Dimension) /\ member-of(Dhierarchy, y) /\ c-inst(z, Dhierarchy) → conformance(x, z)* |
| (b) | | *∀(x, y) ∃(z) c-inst(x, NOhierarchy) /\ member-of(NOparent, x) /\ c-inst(y, NOparent) /\ ConformanceHierarchy(z, x) /\ member-of(Hparent, z) /\ c-inst(y, Hparent) → non-onto parent(y, x)* |
| (c) | | *∀(x, y) ∃(z) c-inst(x, NOhierarchy) /\ member-of(NOchild, x) /\ c-inst(y, NOchild) /\ ConformanceHierarchy(z, x) /\ member-of(Hchild, z) /\ c-inst(y, Hchild) → non-onto child(y, x)* |
| AC10 (a) | Non-strict hierarchy | *∀(x, y) ∃(z) c-inst(x, NShierarchy) /\ c-inst(y, Dimension) /\ member-of(Dhierarchy, y) /\ c-inst(z, Dhierarchy) → conformance(x, z)* |
| (b) | | *∀(x, y) ∃(z) c-inst(x, NShierarchy) /\ member-of(Nsparent, x) /\ c-inst(y, NSparent) /\ ConformanceHierarchy(z, x) /\ member-of(Hparent, z) /\ c-inst(y, Hparent) → non-strict parent(y, x)* |
| (c) | | *∀(x, y) ∃(z) c-inst(x, NShierarchy) /\ member-of(NSchild, x) /\ c-inst(y, NSchild) /\ ConformanceHierarchy(z, x) /\ member-of(Hchild, z) /\ c-inst(y, Hchild) → non-strict child(y, x)* |

mapping of instances for the dimension levels in a non-covering hierarchy because we assume that the non-covering hierarchies between the levels are for a limited number of instances; an alternative approach of creating a new hierarchy between the levels is redundant and complicates the schema at the logical level. A hierarchy instance consists of an instance of a parent level, an instance of child level, and a conformance relationship to a hierarchy (AC8e). The conformance relationship maps the hierarchy

instance to a hierarchy construct. The *NChierarchy* construct represents a non-covering hierarchy between two levels along a path in the dimension. It consists of a parent level, a child level and a conformance relation to a path. The path represents a set of hierarchies in a dimension. The parent and child level in the non-covering hierarchy have a conformance relationship to a dimension path such that the parent level is an ancestor of the child level and not a direct parent (AC8a).

A *non-onto hierarchy* occurs when an instance of a parent level does not have an instance of the child level to drill-down to. Non-onto hierarchies are typically supported by allowing the child level in a hierarchy to have null values [31, 35, 36]. We allow a direct hierarchy to change to a non-onto hierarchy. Similarly to a non-covering hierarchy, a non-onto hierarchy has a conformance relationship (AC9) and an instance mapping between parent and child (not shown here).

According to a strict hierarchy, there is a one-to-many cardinality relationship between a parent and child level in a hierarchy. A *non-strict hierarchy* occurs when there is a many-to-many relationship between the parent and child level in a hierarchy. Pedersen et al. survey data models for non-strict hierarchy support and conclude that none of the proposed models at that time supported non-strict hierarchy [37]. Some researchers mention the problems caused by many-to-many cardinality such as aggregating the measure twice [2, 14, 40] but do not provide the solution to support it. Malinowski and Zimanyi propose modeling non-strict hierarchy [31] in two different ways: (1) a bridge table [27], and (2) mapping non-strict hierarchies to strict hierarchies [37]. In our model we implement a bridge table to support aggregation over a non-strict hierarchy. A detailed model for maintaining correct aggregation during schema evolution has been developed but is beyond the scope of this paper.

**Table 4.** Deletion Constraints (Extended Hierarchy)

| ID | Construct | Constraint |
|---|---|---|
| DC5 | Path | $\forall$ *(a, b) c-inst(a, Dimension)* $\wedge$ *member-of(Dhierarchy, a)* $\wedge$ *member-of(Dpath, a)* $\wedge$ $\neg$*c-inst(b, Dhierarchy)* $\rightarrow$ *delete(b, Dpath)* |
| DC6 | Non-covering hierarchy | $\forall$*(a, b, c)* $\exists$*(x) c-inst(a, NChierarchy)* $\wedge$ *member-of(NCchild, a)* $\wedge$ *member-of(NCparent, a)* $\wedge$ *c-inst(b, NCparent)* $\wedge$ *c-inst(c, NCchild)* $\wedge$ *c-inst(x, Hierarchy)* $\wedge$ *member-of(Hparent, x)* $\wedge$ *member-of(Hchild, x)* $\wedge$ *c-inst(b, Hparent)* $\wedge$ *c-inst(c, Hchild)* $\rightarrow$ *delete(a, NChierarchy)* |
| DC7 | Non-onto hierarchy | $\forall$*(a,* $h_i$*) c-inst(a, NOhierarchy)* $\wedge$ *conformance-of(*$h_i$*, a)* $\wedge$ $\neg$*c-inst(*$h_i$*, DHierarchy)* $\rightarrow$ *delete(a, NOhierarchy)* |
| DC8 | Non-strict hierarchy | $\forall$*(a,* $h_i$*) c-inst(a, NShierarchy)* $\wedge$ *conformance-of(*$h_i$*, a)* $\wedge$ $\neg$*c-inst(*$h_i$*, DHierarchy)* $\rightarrow$ *delete(a, NShierarchy)* |

In the next section, a multilevel dictionary definition is given to illustrate the meta-constructs and constructs of our core model along with an example schema.

## 4 Multilevel Dictionary Implementation

Atzeni, Cappellari, and Bernstein [1] describe an implementation of a multilevel dictionary to manage schemas and describe models of interest. Their approach allows

model-independent schema translation because all models are described with metaconstructs in a supermodel that generalizes all other models. They produce an XML schema representation of an Entity-Relationship schema as an example of the kind of model-independent reporting supported by their approach. They contribute a visible description of models and specification of translations. We utilize the same approach here for representing the core features of a data warehouse, although we use the metaconstructs of ULD rather than the exact supermodel given by Atzeni et al. [1]. Note that both the ULD and the multilevel dictionary approaches model transformation rules outside of their structural formalisms; they use Datalog rules that are independent of the engine used to interpret them. This section details our structural description.

### 4.1 An Example Schema

An example of a data warehouse schema is shown in Figure 2 using DFM notation [18] and our extensions (Figure 1). Our metamodel, corresponding to the ULD constructs in Section 2, is shown in Table 5. The model is represented in a tabular format similar to the multilevel dictionary method. As an example, the construct *c4* has a $set_{ct}$ construct type and represents the information that *FactSet* is a *set-of Fact*.

A multilevel dictionary representation of *Schema* is given in Table 6. The representation includes some set-valued attributes to ease understanding of the definition in a top-down fashion; for implementation in a relational DBMS (Section 4.2), the tables are normalized. The schema has a unique name, *SalesSchema*. The *FactSet* includes the fact *Sales* and the *DimensionSet* is a set of all the dimensions in the schema. The records in the column *FactSet* and *DimensionSet* have a foreign key reference to the *Fact* and *Dimension* construct tables.

The multilevel dictionary representations of *Fact*, names of primary key fields, and measures are listed in Tables 7, 8, and 9, respectively. For example, *f1* is a fact with a unique name *Sales*, primary key fields *Product#, Time#*, and *Location#*, and measures *m1, m2,* and *m3* that represent *Qty Sold, Revenue,* and *No. Customer,* respectively.

Table 10 gives a multilevel dictionary representation of a *Dimension* construct. A *Dimension* is described by a unique name, a set of levels, a primary key, and sets of hierarchies and paths over the levels. For example, the *Location* dimension has the levels *L19*, *L20,* and *L21* representing *City, Province,* and *Country*. The primary key for the dimension is *Location#*, and *h22* and *h23* are the hierarchies of the dimension. An empty record in the *HierarchySet* indicates the lack of a hierarchy among the levels of a dimension. Table 11 lists the multilevel dictionary representation for *Level* construct.

A *Hierarchy* construct is defined by a parent level and a child level. The parent level drills-down to a child level. For example, levels *L8* and *L7* are parent and child level, respectively, and represent the information *Year* drills-down to *Quarter*. Table 12 lists the multi-level dictionary representation for the *Hierarchy* construct.

Table 13 lists multiple paths that occur in dimensions. For example, *p4* is the path *Item, Category, Corporation* in the dimension *Product*. Non-covering hierarchies, such as that between *District* and *Province* in the *Location* dimension (*nc3*) are given in Table 14, along with the path the hierarchy conforms to (*p9*). Table 15 records the non-onto hierarchy between *City* and *IP Add*, while Table 16 records the non-strict relationships between levels, such as *Province* and *Cell*.

**Table 5.** Metamodel: Constructs and Construct Types

| Schema | |
|---|---|
| OID | Name |
| S1 | DW Schema |

| Construct Type | | |
|---|---|---|
| OID | Name | *Schema* |
| ct1 | struct-ct | S1 |
| ct2 | set-ct | S1 |
| ct3 | union-ct | S1 |
| ct4 | atomic-ct | S1 |
| ct5 | predefined | S1 |

| Constructs | | | |
|---|---|---|---|
| OID | Name | *Construct Type* | Value |
| c1 | Schema | ct1 | c2, c6 |
| c2 | FactSet | ct2 | c3 |
| c3 | Fact | ct1 | c4, c10, c17 |
| c4 | FactPKeySet | ct2 | c5 |
| c5 | FactPkey | ct1 | c18 |
| c6 | DimensionSet | ct2 | c7 |
| c7 | Dimension | ct1 | c8, c12, c15 |
| c8 | LevelSet | ct2 | c9 |
| c9 | Level | ct1 | c18 |
| c10 | MeasureSet | ct2 | c11 |
| c11 | Measure | ct1 | c18 |
| c12 | HierarchySet | ct2 | c13 |
| c13 | Hierarchy | ct1 | c9, c9 |
| c15 | DimensionPkey | ct1 | c18 |
| c16 | Cube | ct1 | c3, c6 |
| c17 | AttributeSet | ct2 | c18 |
| c18 | Attribute | ct5 | |
| c19 | PathSet | ct2 | c20 |
| c20 | Path | ct2 | c21 |
| c21 | PathHierarchy | ct1 | c13 |
| c22 | NChierarchy | ct1 | c9, c9, c21 |
| c23 | NOhierarcy | ct1 | c9, c9, c13 |
| c24 | NShierarcy | ct1 | c9, c9, c13 |

**Table 6.** Model: Schema Definition

| Schema | | | | |
|---|---|---|---|---|
| OID | Name | *Construct Name* | *FactSet* | *DimensionSet* |
| s1 | SalesSchema | c1 | f1 | d1, d2, d3 |

**Table 7.** Fact Construct

| Fact | | | | |
|---|---|---|---|---|
| OID | FName | *Construct Name* | *MeasureSet* | *Pkey* |
| f1 | Sale | c3 | m1, m2, m3 | pk1, pk2, pk3 |

**Table 8.** Primary Keys of the Fact

| FactPrimaryKey | | |
|---|---|---|
| OID | Name | *Construct Name* |
| pk1 | Time# | c5 |
| pk2 | Product# | c5 |
| Pk3 | Location# | c5 |

**Table 9.** Measure Construct

| Measure | | |
|---|---|---|
| OID | Name | *Construct Name* |
| m1 | Qty Sold | c11 |
| m2 | Revenue | c11 |
| m3 | No. Customer | c11 |

**Table 10.** Dimension Construct

| Dimension | | | | | | |
|---|---|---|---|---|---|---|
| OID | DName | *Construct Name* | *DLevel* | *DPkey* | *DHierarchy* | *Dpath* |
| d1 | Time | c7 | L1, L2, L3, L4, L5, L6, L7, L8 | pk1 | h1, h2, h3, h4, h5, h6, h7 | p1, p2 |
| d2 | Product | c7 | L9, L10, L11, L12, L13 | pk2 | h8, h9, h10, h11, h12 | p3, p4 |
| d3 | Location | c7 | L14, L15, L16, L17, L18, L19, L20, L21 | pk3 | h13, h14, h15, h16, h17, h18, h19, h20, h21, h22, h23 | p5, p6, p7, p8, p9 |

**Table 11.** Level Construct

| Level | | |
|---|---|---|
| OID | Name | *Construct Name* |
| L1 | Second | c9 |
| L2 | Minute | c9 |
| L3 | Hour | c9 |
| L4 | Day | c9 |
| L5 | Week | c9 |
| L6 | Month | c9 |
| L7 | Quarter | c9 |
| L8 | Year | c9 |
| L9 | Item | c9 |
| L10 | Brand | c9 |
| L11 | Company | c9 |
| L12 | Category | c9 |
| L13 | Corporation | c9 |
| L14 | Co-ordinate | c9 |
| L15 | Street | c9 |
| L16 | District | c9 |
| L17 | IP add | c9 |
| L18 | Cell | c9 |
| L19 | City | c9 |
| L20 | Province | c9 |
| L21 | Country | c9 |

**Table 12.** Hierarchy Construct

| Hierarchy | | |
|---|---|---|
| OID | *ParentLevel* | *ChildLevel* |
| h1 | L2 | L1 |
| h2 | L3 | L2 |
| h3 | L4 | L3 |
| h4 | L5 | L4 |
| h5 | L6 | L4 |
| h6 | L7 | L6 |
| h7 | L8 | L7 |
| h8 | L10 | L9 |
| h9 | L11 | L10 |
| h10 | L12 | L9 |
| h11 | L13 | L11 |
| h12 | L13 | L12 |
| h13 | L17 | L14 |
| h14 | L18 | L14 |
| h15 | L15 | L14 |
| h16 | L16 | L15 |
| h17 | L19 | L17 |
| h18 | L19 | L18 |
| h19 | L19 | L16 |
| h20 | L20 | L17 |
| h21 | L20 | L18 |
| h22 | L20 | L19 |
| h23 | L21 | L20 |

### 4.2 Schema Evolution Implementation

In this section, an overview of the MDD implementation of the formal model is given along with the algorithm for a schema evolution operator. An example of schema evolution is given that uses the schema in Figure 2. All user interaction is conducted within the Microsoft SQL Server 2000 environment. A user may either use a simple GUI to execute a stored procedure where he or she will be prompted to supply the

arguments or may use the SQL command line. The procedure then updates records in the tables (such as those presented in Section 4.1) that may in turn cause triggers to execute on the tables. Example screenshots of a trigger and a stored procedure are given in Figures 5 and 6, respectively.

**Table 13.** Path Construct

| Path | | |
|---|---|---|
| OID | *Phierarchy* | *Construct Name* |
| p1 | h1, h2, h3, h4 | c20 |
| p2 | h1, h2, h3, h5, h6, h7 | c20 |
| p3 | h8, h9, h11 | c20 |
| p4 | h10, h12 | c20 |
| p5 | h13, h17, h22, h23 | c20 |
| p6 | h13, h20, h23 | c20 |
| p7 | h14, h18, h22, h23 | c20 |
| p8 | h14, h21, h23 | c20 |
| p9 | h15, h16, h19, h22, h23 | c20 |

**Table 14.** Non-Covering Construct.

| NChierarchy | | | |
|---|---|---|---|
| OID | *NCparent* | *NCchild* | *Conformance* |
| Nc1 | L16 | L14 | p9 |
| Nc2 | L19 | L14 | p9 |
| Nc3 | L20 | L16 | p9 |

**Table 15.** Non-Onto Construct.

| NOhierarcy | | |
|---|---|---|
| OID | *NOparent* | *Conformance* |
| No1 | L19 | h17 |

**Table 16.** Non-Strict Construct.

| NShierarchy | | | |
|---|---|---|---|
| OID | *NCparent* | *NCchild* | *Conformance* |
| ns1 | L19 | L18 | h18 |
| ns2 | L20 | L18 | h21 |

The MDD definition for the ULD constructs such as facts, dimensions, levels, and hierarchies are implemented using tables in Microsoft SQL Server 2000 as shown in Figure 4. Enforcement of the semantics is accomplished by triggers over the tables that implement addition and deletion constraints over the features. A database trigger is procedural code that is automatically executed in response to certain events on a particular table in a database. The events that cause a trigger to execute are insert, update, and delete. For example, when a hierarchy is added to the *DHierarchySet* table, the child level of a hierarchy has to be an existing level in the dimension. The

trigger ensures that when a hierarchy is added to *DHierarchySet* the child level in the hierarchy is a level in *DLevelSet*. A trigger is created over the *DHierarchySet* and is executed when a record is added or updated in the table. If a new record is inserted in this table and if the child level is not present in the table *DLevelSet*, the transaction is rolled back. We implement 23 addition constraints (Tables 1 and 3) and 12 deletion constraints (Tables 2 and 4) as triggers for enforcing correct semantics for schema evolution. A sample trigger is shown in the screenshot in Figure 5 for deletion of a non-onto hierarchy.

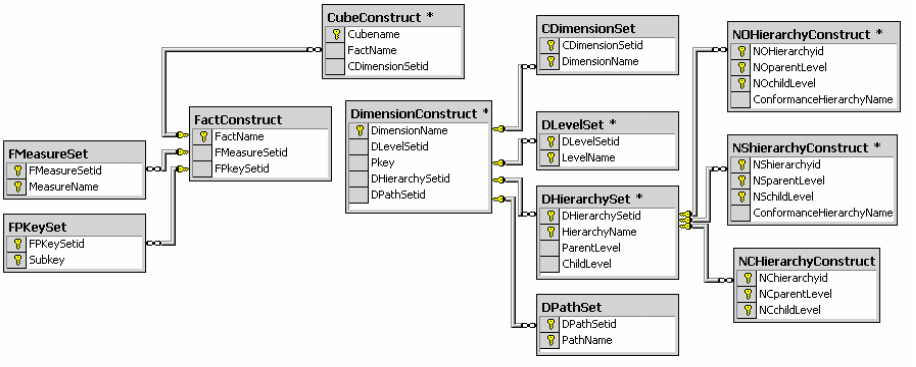


**Fig. 4.** MDD Implementation

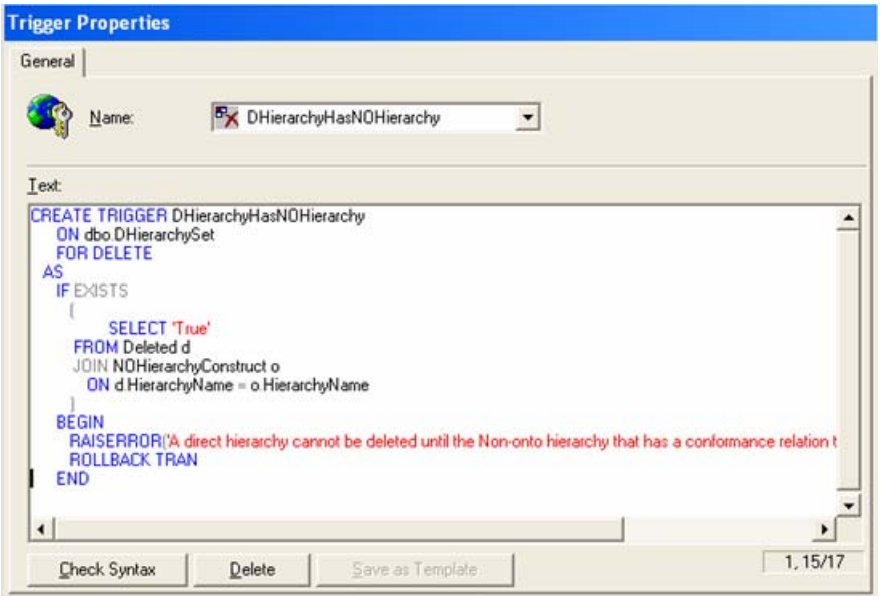


**Fig. 5.** Screenshot of a Trigger

The records in the tables represent a schema and the addition, deletion, and update of records represent the changes to a schema (i.e., schema evolution.) In our tool, the evolution operators have been implemented as stored procedures that accept arguments from the user and accordingly make changes to a schema. The changes rely on triggers to enforce semantics. A stored procedure is a subroutine written in T-SQL code to access the SQL Server relational database system. The tool supports typical evolution operators over the core features of a data warehouse [8, 9, 10, 12, 13, 17, 25, 28, 39] such as add a new level to a dimension, add a hierarchy between two levels of a dimension, delete a level from a dimension, and add a dimension to fact, as well as add/delete multiple, non-strict, non-onto, and non-covering hierarchies [22, 31, 35, 36, 44]. Our operators subsume those proposed in the literature and additional capabilities for addition and deletion of extended hierarchies. We implement these operators as 23 stored procedures, described in Table 17 along with their arguments (i.e., tables.) An example of a stored procedure is shown in the screenshot given in Figure 6 for adding a non-covering hierarchy.

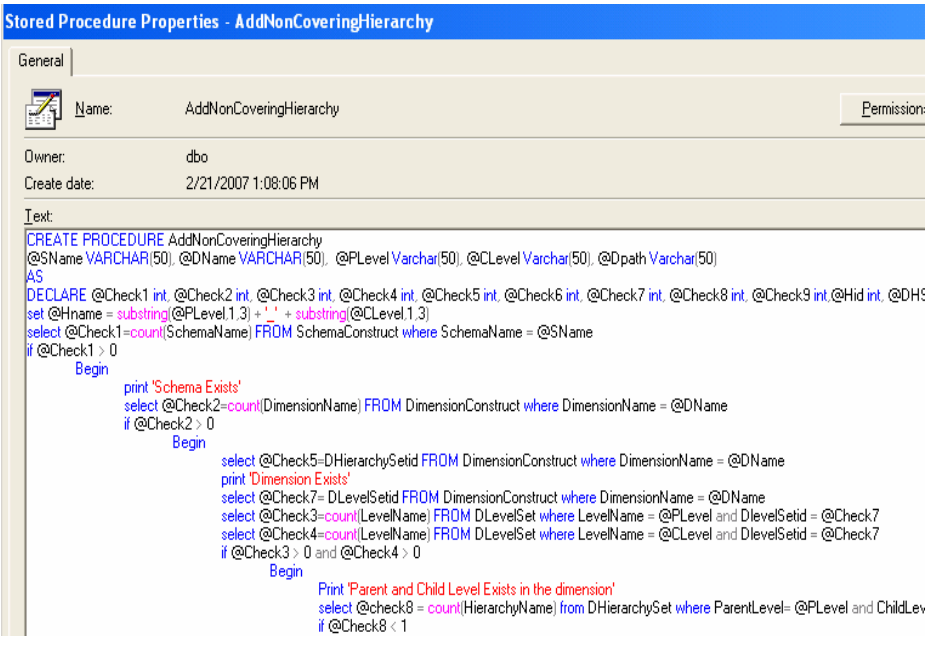


**Fig. 6.** Screenshot of a Stored Procedure Excerpt

An example of an evolution operator is *AddNonCoveringHierarchy* that creates a non-covering hierarchy relationship between two levels of a dimension. The arguments passed are schema name, dimension, child level, and parent level. The arguments passed by the evolution operator are added to the tables only if they are consistent with the addition constraints of *NChierarcyConstruct*. This ensures the correctness of schema evolution. The stored procedure that implements the *AddNonCoveringHierarchy* operator is shown in Figure 6 and an algorithm for it is given in Figure 7. The evolution operator to delete a non-covering hierarchy checks that the participating levels conform to some path that remains valid after the non-covering, ancestor-child relationship is removed.

**Table 17.** Evolution Operators as Stored Procedures

| Evolution Operator | Tables Affected | Description |
|---|---|---|
| AddSchema | schema | Creates a schema to which facts and dimensions can be added |
| AddFactToSchema | schema, fact, measures, primary key | Adds a fact to an existing schema |
| AddDimensionToSchema | schema, dimension, primary key | Adds a dimension to an existing schema |
| AddLevelToDimension | schema, dimension, level | Adds a level in a existing dimension |
| AddHierarchyToDimension | schema, dimension, parent level, child level | Adds a hierarchy between two existing levels of a dimension |
| AddCubeToSchema | schema, cube, fact, set of dimension | Creates a cube by joining a set of existing dimensions to a fact |
| AddDimensionToFactInCube | schema, cube, fact, dimension | Connects a dimension to a fact in a existing cube |
| DeleteDimensionFactInCube | schema, cube, fact, dimension | Deletes a dimension from a fact in a existing cube |
| DeleteCubeFromSchema | schema, cube | Deletes a cube in an existing schema |
| DeleteFactFromSchema | schema, fact | Deletes a fact from an existing schema |
| DeleteHierarchyInDimension | schema, dimension, parent level, child level | Deletes a hierarchy in a dimension |
| DeleteLevelInDimension | schema, dimension, level | Deletes a level in a existing dimension |
| DeleteDimensionInSchema | schema, dimension | Deletes a dimension from an existing schema |
| DeleteSchema | schema | Deletes a schema |
| RenameFactInSchema | schema, fact | Rename a fact in an existing schema |
| RenameDimensionInSchema | schema, dimension | Rename a dimension in an existing schema |
| RenameLevelInDimension | schema, dimension, level | Rename a level in an dimension |
| AddNonStrictHierarchy | schema, dimension, parent level, child level | Adds a non-strict hierarchy in a dimension |
| DeleteNonStrictHierarchy | schema, dimension, parent level, child level | Deletes a non-strict hierarchy in a dimension |
| AddNonOntoHierarchy | schema, dimension, parent level, child level | Adds a non-onto hierarchy in a dimension |
| DeleteNonOntoHierarchy | schema, dimension, parent level, child level | Deletes a non-onto hierarchy in a dimension |
| AddNonCoveringHierarchy | schema, dimension, parent level, child level | Adds a non-covering hierarchy in a dimension |
| DeleteNonCoveringHierarchy | schema, dimension, parent level, child level | Deletes a non-covering hierarchy in a dimension |
| AddMultipleHierarchyToDimension | schema, dimension, parent level, child level | Adds a multiple hierarchy in a dimension |
| ReExaminePath | schema, dimension | Modifies the path as the hierarchy changes |

Consider a scenario where there is an existing schema with a dimension *Location* that has the levels *Country*, *Province* and *City* and the hierarchies such that *Country* drills-down to *Province* and *Province* drills-down to *City*. The *AddNon-CoveringHierarchy* operator is used to create a non-covering hierarchy between the levels *Country* and *City*. The hierarchy is successfully created since it satisfies Steps 1 to 5 in the algorithm shown in Figure 7. Consider a second scenario when the operation is to add a non-covering hierarchy between *Province* and *City.* In this case, the schema evolution operation is unsuccessful in creating the hierarchy as it does not satisfy the constraint that a parent and child level of a non-covering hierarchy cannot be a direct hierarchy (AC8d). In other words, the child must roll-up to an ancestor, not a direct parent.

---

Algorithm AddNonCoveringHierarchy
**Input**: Schema Name, Dimension Name, Child Level, Parent Level
**Output**: A non-covering hierarchy is created between the parent and child levels of a dimension

Step 1: Check if Schema Name is valid.
Step 2: Check if Dimension Name is a dimension in the schema.
Step 3: Check if Parent and Child Level are existing levels of the dimension.
Step 4: Check if Parent Level is an ancestor (not a direct parent) of Child Level along a path in the dimension.
Step 5: If Steps 1 to 4 are satisfied, add Parent and Child Levels in the NCHierarchy table.

---

**Fig. 7.** Evolution Operator Example

**Table 18.** Schema Evolution on *Location* Dimension [26]

| Operator | Arguments | Constraints |
|---|---|---|
| AddMultipleHierarchy | SalesSchema, Location, Province, IP add | AC5, AC7 |
| AddMultipleHierarchy | SalesSchema, Location, City, IP add | AC5, AC7 |
| AddMultipleHierarchy | SalesSchema, Location, IP add, Co-ordinate | AC5, AC7 |
| AddMultipleHierarchy | SalesSchema, Location, Province, Cell | AC5, AC7 |
| AddMultipleHierarchy | SalesSchema, Location, City, Cell | AC5, AC7 |
| AddMultipleHierarchy | SalesSchema, Location, Cell, Co-ordinate | AC5, AC7 |
| AddNonCoveringHierarchy | SalesSchema, Location, Province, District | AC8 |
| AddNonCoveringHierarchy | SalesSchema, Location, District, Co-ordinate | AC8 |
| AddNonCoveringHierarchy | SalesSchema, Location, City, Co-ordinate | AC8 |
| AddNonOntoHierarchy | SalesSchema, Location, City, IP add | AC9 |
| AddNonStrictHierarchy | SalesSchema, Location, Province, Cell | AC10 |
| AddNonStrictHierarchy | SalesSchema, Location, City, Cell | AC10 |
| AddNonStrictHierarchy | SalesSchema, Location, District, Street | AC10 |
| PathFinder | SalesSchema, Location | AC7 |

In order to show schema evolution via adding constructs to a schema, we assume that a schema called *SalesSchema* with a fact called *Sale* and measures shown in Figure 2 has been created. We assume that the direct hierarchies in the *Location* dimension have already been created. The operations needed to construct the extended

hierarchies of the *Location* dimension (as in the final result shown in Figure 2) are given in Table 18. In this example of a complex evolution task, at least one example of every kind of extended hierarchy semantics is illustrated.

Our tool provides a methodology to define a data warehouse schema and explore evolution over core and additional features. Users can use the tool to create semantically correct data warehouse schemas and populate them. The evolution operators can be used to make changes to a schema by modifying the tables representing the metadata about the schema. The triggers ensure the correctness and consistency of a schema and help a user to visualize the process of schema design and evolution. Most importantly, the semantics of advanced features for design and evolution of schemas can be explored experientially and there is no other tool that supports this kind of activity.

## 5 Conclusions and Future Work

We introduce a formal model of core features and specify semantics of schema evolution operators for a data warehouse. We contribute a tool that implements schema evolution operators as stored procedures that invoke triggers to enforce schema correctness. Our tool allows a user to create, populate, and query over a basic data warehouse schema without requiring any knowledge more specialized than the ability to use a relational database to operate the tool. In addition, no other system supports investigating the impact of schema evolution over both core and advanced data warehouse schema modeling constructs.

Our research provides a basis for further investigation of generic tools for schema and model management such as browsing [5, 6, 7] and reporting and visualization [1]. Because MDD definitions exist for ER and OODB models and ULD definitions exist for XML, RDF, and relational models, information interchange can be achieved by defining mappings between these models and ours. Automating the generation of triggers and stored procedures from rules specified in Datalog or first-order logic also remains as a topic for future investigation.

In our work in progress, we have defined multidimensional instance lattices in order to enforce correct semantics of schema evolution over data instances. We have experimented with our tool using schema evolution modeled by others [17] as well as a variety of test cases since there are no schema evolution examples in the literature using extended hierarchy semantics. A complete implementation for data migration between evolved schemas remains as future work.

We are currently implementing model management capabilities with our tool. We have added a front-end tool that utilizes StarER [43] as a data warehouse conceptual model, represents it in our MDD logical model, and produces a Microsoft SQL Server Analysis Services implementation. We are currently incorporating other conceptual data warehouse models (ME/R [42] and DFM [18]) at the front-end along with algorithms to support schema merging over the metamodel defined here. Tools for translating conceptual schemas have been proposed [20, 21, 29, 38] but none use a multilevel data dictionary (*model gen*) approach as we do here; our approach is intended to allow interoperability among heterogeneous models rather to support a specific data model. We focus on the relational model as our implementation model

vehicle here since our structures and rules translate naturally to triggers, and our operators translate naturally to stored procedures. The target model (a ROLAP implementation) could be replaced with a MOLAP implementation in the future.

Alternative platform-independent approaches to data warehouse schema specification could be explored as the basis for schema evolution; Mazón et al. [32, 34] propose a model-driven architecture approach to specify conceptual schemas and cube metadata for supporting OLAP applications. Query-view-transformation (QVT) mappings could potentially be used for specifying evolution operators.

Future work also includes investigating how the model can be integrated with other related research such as versioning and cross-version querying [17, 41, 47] and ETL process modeling [46]. Using ETL conceptual modeling to specify data migration paths between versions to facilitate on-demand query processing for other versions appears to be a promising research direction. Another interesting approach proposed for maintaining an information system containing a relational database considers the correctness of queries when database evolution occurs [33]; investigating whether or how to extend the paradigm to accommodate multidimensional modeling is an open topic that could leverage the foundation we provide here. Similarly, Kaas et al. [28] investigate the impact of schema evolution on star and snowflake schemas for browse and aggregation queries. It would be interesting to explore the impact of additional evolution operators over extended semantics and create a framework for rewriting queries based on the operators.

## References

[1] Atzeni, P., Cappellari, P., Bernstein, P.: A multilevel dictionary for model management. In: Delcambre, L.M.L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, Ó. (eds.) ER 2005. LNCS, vol. 3716, pp. 160–175. Springer, Heidelberg (2005)

[2] Agrawal, R., Gupta, A., Sarawagi, S.: Modeling Multidimensional Databases. In: Proceedings of the 13th International Conference on Data Engineering (ICDE), Birmingham, U.K, April 7-11, 1997, pp. 232–243 (1997)

[3] Abelló, A., Samos, J., Saltor, F.: YAM2 (Yet Another Multidimensional Model). In: Proceedings of the International Database Engineering & Applications Symposium (IDEAS), Edmonton, Canada, July 17-19, 2002, pp. 172–181 (2002)

[4] Bækgaard, L.: Event-Entity-Relationship Modeling in Data Warehouse Environments. In: Proceedings of 2nd ACM Second International Workshop on Data Warehousing and OLAP (DOLAP), Kansas City, Missouri, USA, November 6, 1999, pp. 9–14 (1999)

[5] Bowers, S., Delcambre, L.: On Modeling Conformance for Flexible Transformation over Data Models. Knowledge Transformation for the Semantic Web 95, 34–48 (2003)

[6] Bowers, S., Delcambre, L.: The uni-level description: A uniform framework for representing information in multiple data models. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 45–58. Springer, Heidelberg (2003)

[7] Bowers, S., Delcambre, L.: Using the Uni-Level Description (ULD) to Support Data-Model Interoperability. Data and Knowledge Engineering 59(3), 511–533 (2006)

[8] Bouzeghoub, M., Kedad, Z.: A logical model for data warehouse design and evolution. In: Kambayashi, Y., Mohania, M., Tjoa, A.M. (eds.) DaWaK 2000. LNCS, vol. 1874, pp. 178–188. Springer, Heidelberg (2000)

[9] Blaschka, M., Sapia, C., Höfling, G.: On schema evolution in multidimensional databases. In: Mohania, M., Tjoa, A.M. (eds.) DaWaK 1999. LNCS, vol. 1676, pp. 153–164. Springer, Heidelberg (1999)
[10] Chen, J., Chen, S., Rundensteiner, E.: A transactional model for data warehouse maintenance. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, pp. 247–262. Springer, Heidelberg (2002)
[11] Chaudhuri, S., Dayal, U.: An Overview of Data Warehousing and OLAP Technology. SIGMOD Record 26(1), 65–74 (1997)
[12] Claypool, K., Natarajan, C., Rundensteiner, E., Rundensteiner, E.: Optimizing Performance of Schema Evolution Sequences. In: Dittrich, K.R., Guerrini, G., Merlo, I., Oliva, M., Rodriguez, M.E. (eds.) ECOOP-WS 2000. LNCS, vol. 1944, pp. 114–127. Springer, Heidelberg (2001)
[13] Claypool, K., Rundensteiner, E., Heineman, G.: Evolving the Software of a Schema Evolution System. In: Balsters, H., De Brock, B., Conrad, S. (eds.) FoMLaDO 2000 and DEMM 2000. LNCS, vol. 2065, pp. 68–84. Springer, Heidelberg (2001)
[14] Datta, A., Thomas, H.: A Conceptual Model and Algebra for On-line Analytical Processing in Data Warehouses. In: Proceedings of the 7th Workshop for Information Technology and Systems (WITS), Atlanta, Georgia, USA, December 13-14, 1997, pp. 91–100 (1997)
[15] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Datacube: A Relational Aggregation Operator Generalizing Group-by, Cross-tab, and Sub-totals. Journal of Data Mining and Knowledge Discovery, ch. 1, 29–53 (1997)
[16] Gyssens, M., Lakshmanan, L.: A Foundation for Multi-Dimensional Databases. In: Proceedings of the 23rd International Conference on Very Large Databases (VLDB), Athens, Greece, August 25-29, 1997, pp. 106–115 (1997)
[17] Golfarelli, M., Lechtenbörger, J., Rizzi, S., Vossen, G.: Schema Versioning in Data Warehouses. In: Wang, S., Tanaka, K., Zhou, S., Ling, T.-W., Guan, J., Yang, D.-q., Grandi, F., Mangina, E.E., Song, I.-Y., Mayr, H.C. (eds.) ER Workshops 2004. LNCS, vol. 3289, pp. 415–428. Springer, Heidelberg (2004)
[18] Golfarelli, M., Maio, D., Rizzi, S.: The Dimensional Fact Model: A Conceptual Model for Data Warehouses. International Journal of Cooperative Information Systems (IJCIS) 7(2-3), 215–247 (1998)
[19] Golfarelli, M., Rizzi, S.: A Methodological Framework for Data Warehousing Design. In: Proceedings of the 1st International Workshop on Data Warehousing and OLAP (DOLAP), Washington, DC, USA, November 2-7, pp. 3–9 (1998)
[20] Golfarelli, M., Rizzi, S.: WAND: A CASE Tool for Data Warehouse Design. In: Proceedings of the 17th International Conference on Data Engineering (ICDE), Heidelberg, Germany, April 2-6, pp. 7–9 (2001)
[21] Hahn, K., Sapia, C., Blaschka, M.: Automatically Generating OLAP Schemata from Conceptual Graphical Models. In: Proceedings of the 3rd ACM International Workshop on Data Warehousing and OLAP (DOLAP), pp. 9–16 (2000)
[22] Hümmer, W., Lehner, W., Bauer, A., Schlesinger, L.: A decathlon in multidimensional modeling: Open issues and some solutions. In: Kambayashi, Y., Winiwarter, W., Arikawa, M. (eds.) DaWaK 2002. LNCS, vol. 2454, pp. 275–285. Springer, Heidelberg (2002)
[23] Hüsemann, B., Lechtenbörger, J., Vossen, G.: Conceptual Data Warehouse Design. In: Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW), Stockholm, Sweden, June 5-6, pp. 6:1–6:11(2000)

[24] Hurtado, C., Mendelzon, A.: Reasoning about summarizability in heterogeneous multidimensional schemas. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 375–389. Springer, Heidelberg (2000)
[25] Hurtado, C., Mendelzon, A., Vaisman, A.: Maintaining Data Cubes under Dimension Updates. In: Proceedings of 15th International Conference of Data Engineering (ICDE), Sydney, Australia, March 23-26, pp. 346–355 (1999)
[26] Jensen, C., Kligys, A., Pedersen, T., Timko, I.: Multidimensional Data Modeling for Location Based Services. The VLDB Journal 13(1), 1–21 (2004)
[27] Kimball, R.: The Data Warehouse Toolkit. John Wiley & Sons, Inc., New York (1996)
[28] Kaas, C., Pedersen, T.B., Rasmussen, B.: Schema Evolution for Stars and Snowflakes. In: Proceedings of the 6th International Conference on Enterprise Information Systems, Porto, Portugal, April 14-17, pp. 425–433 (2004)
[29] Luján-Mora, S., Trujillo, J., Song, I.: A UML Profile for Multidimensional Modeling in Data Warehouses. Data and Knowledge Engineering 59(3), 725–769 (2006)
[30] Li, C., Wang, X.: A Data Model for Supporting On-Line Analytical Processing. In: Proceedings of the 5th International Conference on Information and Knowledge Management (CIKM), Rockville, Maryland, November 12-16, pp. 81–88 (1996)
[31] Malinowski, E., Zimányi, E.: OLAP hierarchies: A conceptual perspective. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 477–491. Springer, Heidelberg (2004)
[32] Mazón, J.-N., Trujillo, J.: An MDA Approach for the Development of Data Warehouses. Decision Support Systems 45(1), 41–58 (2008)
[33] Papastefanatos, G., Vassiliadis, P., Vassiliou, Y.: Adaptive Query Formulation to Handle Database Evolution. In: Proceedings of the Conference on Advanced Information Systems Engineering: CAiSE Forum, Luxembourg (2006)
[34] Pardillo, J., Mazón, J.-N., Trujillo, J.: Model-driven Metadata for OLAP Cubes from the Conceptual Modeling of Data Warehouses. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2008. LNCS, vol. 5182, pp. 13–22. Springer, Heidelberg (2008)
[35] Pedersen, T., Jensen, C.: Research Issues in Clinical Data Warehousing. In: Proceedings of the 10th International Conference on Scientific and Statistical Database Management, Capri, Italy, July 1-3, pp. 43–52 (1998)
[36] Pedersen, T., Jensen, C.: Multidimensional Data Modeling for Complex Data. In: Proceedings of 15th International Conference on Data Engineering (ICDE), Sydney, Australia, March 23-26, pp. 336–345 (1999)
[37] Pedersen, T., Jensen, C., Dyreson, C.: A Foundation for Capturing and Querying Complex Multidimensional Data. Information Systems 26(5), 383–423 (2001)
[38] Prat, N., Akoka, J., Comyn-Wattiau, I.: A UML-based Data Warehouse Design Method. Decision Support Systems 42, 1449–1473 (2006)
[39] Quix, C.: Repository Support for Data Warehouse Evolution. In: Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW 1999), Heidelberg, Germany, June 14-15, p. 4 (1999)
[40] Rafanelli, M., Shoshani, A.: STORM: A Statistical Object Representation Model. In: Michalewicz, Z. (ed.) SSDBM 1990. LNCS, vol. 420, pp. 14–29. Springer, Heidelberg (1990)
[41] Rizzi, S., Golfarelli, M.: X-Time: Schema Versioning and Cross-Version Querying in Data Warehouses. In: Proceedings of the 23rd International Conference on Data Engineering (ICDE), Istanbul, Turkey, April 15-20, pp. 1471–1472 (2007)
[42] Sapia, C., Blaschka, M., Höfling, G., Dinter, B.: Extending the E/R Model for the Multidimensional Paradigm. In: Kambayashi, Y., Lee, D.-L., Lim, E.-p., Mohania, M., Masunaga, Y. (eds.) ER Workshops 1998. LNCS, vol. 1552, pp. 105–116. Springer, Heidelberg (1999)

[43] Tryfona, N., Busborg, F., Christiansen, J.: StarER: A Conceptual Model for Data Warehouse Design. In: Proceedings of the 2nd ACM International Workshop on Data Warehousing and OLAP, Kansas City, Missouri, USA, November 6, pp. 3–8 (1999)

[44] Tsois, A., Karayannidis, N., Sellis, T.: MAC: Conceptual data modeling for OLAP. In: Proceedings of the 3rd International Workshop on Design and Management of Data Warehouses (DMDW), Interlaken, Switzerland, June 4, p. 5 (2001)

[45] Vassiliadis, P.: Modeling Multidimensional Databases, Cubes and Cube Operations. In: Proceedings of 10th International Conference on Scientific and Statistical Database Management (SSDBM), Capri, Italy, July 1-3, pp. 53–62 (1998)

[46] Vassiliadis, P., Simitsis, A., Georgantas, P., Terrovitis, M., Skiadopoulos, S.: A Generic and Customizable Framework for the Design of ETL Scenarios. Information Systems 30(7), 492–525 (2005)

[47] Wrembel, R., Morzy, T.: Managing and querying versions of multiversion data warehouse. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 1121–1124. Springer, Heidelberg (2006)