FIESTA: A Framework for Schema Evolution in Multidimensional Databases

Markus Blaschka

Institut für Informatik
der Technischen Universität München

FIESTA: A Framework for Schema Evolution in Multidimensional Databases

Dipl.-Inform. Univ. Markus Blaschka

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. E. Jessen
Prüfer der Dissertation:	1. Univ.-Prof. R. Bayer, Ph.D. / University of Illinois, Urbana
	2. Univ.-Prof. Dr. W. Kießling, Universität Augsburg

Die Dissertation wurde am 10. Juli 2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 8. Dezember 2000 angenommen.

To my parents Marion and Bruno Blaschka.

Thanks for all they gave to me.

Acknowledgements

I thank my supervisor, Prof. Rudolf Bayer, Ph.D., for all his support and confidence in this thesis. He helped me a lot with his fruitful comments and ideas. Especially during his sabbatical in Singapore he was very busy with reading the thesis core and our intensive discussions encouraged my own confidence in this work.

Prof. Dr. Ulrike Jaeger awoke my interest for research work during my master's thesis and continued in tracing my work until this thesis found its end. Her support, her enthusiasm, her valuable comments and ideas when proofreading this thesis encouraged me when my motivation began to fade.

A Ph.D. thesis always grows within a project and is to some degree the product of a synergetical environment. I had the luck to work in a great project team, the BabelFish group at FORWISS. Thanks to Dr. Gabriele Höfling and Dr. Barbara Dinter who directed my initial ideas and thoughts to the right research area. Without Carsten Sapia this thesis would not have the same degree of profoundness. Our countless intensive discussions and his always very critical, but extremely useful and constructive comments produced valuable ideas for the overall research contribution of this thesis.
Finally, I would like to thank our master students and interns for all their valuable work. Monika Vetterling implemented the schema evolution component and helped me in refining many ideas of my work. Charlie Hahn designed the concept for our generator component within the BabelFish tool environment and produced numerous valuable results both for the overall project outcome and my work. Thorsten Ulbricht did a good job in examining commercial OLAP products and finding out their peculiarities, weaknesses and strenghts.

Many interesting ideas and discussions arose from the GI[1] working group „Konzepte des Data Warehouse". Here, I experienced numerous helpful discusssions and received immense feedback on my proposed ideas and draft concepts. I still owe a bar of chocolate to Prof. Dr. Alejandro Buchmann and a beer to Dr. Martin Staudt. Thanks to Holger Günzel and Steffen Stock for always defining a clear borderline between our three related approaches.

I thank my colleagues Dr. Volker Markl, Dr. Roland Ritsch, Wolfgang Wohner, and Peter Zoller for extended discussions, valuable ideas, and proofreading this thesis and other publications. Thanks also to my proofreaders Uli Bähr and Charlie Hahn. Many thanks to Prof. Dr. Werner Kießling for being a well-minded co-assessor and giving me valuable feedback.

Finally, I thank my family and friends who helped me relaxing and showed me that life is still more than a PhD thesis. I just name a few here, but thanks to them all. Mrs. and Mr. Emons are perfect housekeepers and prevented my garden from becoming a real jungle. The guys at the gym convinced me that benchpressing is a valuable environment for brainstorming and developing new ideas. Mocca improved my English expression throughout many publications, including this thesis. Since she is not only a good friend but also a world famous barkeeper, she and her cocktails cheered me up in bad times and brought a lot of fun and distraction into my life. Thanks to Martin for all his support during these hard times. He was always there for me whenever I needed him.

Last, but not least, I thank the companies Saeco and Lavazza for keeping the necessary level of caffeine in my body.

[1] German Informatics Society

Zusammenfassung

Neuartige Datenbank-Anwendungen wie Data Warehousing und OLAP (Online Analytical Processing) verwenden zur Beschreibung der Anwendungsdomäne das multidimensionale Datenmodell. OLAP Systeme weisen daher ein multidimensionales Datenbank-Schema auf, um die Anwendungs-Semantik adäquat darzustellen. Mit FIESTA wird eine Methodik zur Schema-Evolution solcher multidimensionaler Schemata vorgestellt. Kern der Arbeit ist eine Schema-Evolutions-Algebra, die eine Formalisierung des multidimensionalen Datenmodells zusammen mit darauf aufbauenden Schema-Evolutions-Operationen beinhaltet. Da OLAP-Systeme meist als Zusatzschichten-Architektur für relationale Datenbanksysteme implementiert werden, wird die Verarbeitung von Sequenzen solcher Schema-Evolutions-Operationen in einem relationalen Datenbanksystem vorgestellt. Dazu wird formal beschreiben, wie ein multidimensionales Schema auf ein entsprechendes relationales DB-Schema abgebildet werden kann. Damit bei dieser Transformation die volle multidimensionale Semantik erhalten bleibt, wird ein entsprechendes Metaschema als Erweiterung des relationalen Systemkatalogs eingeführt. Zur konsistenten Umsetzung von Evolutions-Operationen-Sequenzen erfolgt eine Transformation in entsprechende relationale Evolutions-Kommandos, die neben dem eigentlichen relationalen Datenbankschema auch die Instanzen und die Inhalte des Metaschemas anpassen.

FIESTA wurde im Rahmen einer graphischen Data Warehouse-Entwurfsumgebung prototypisch implementiert. Dabei werden die multidimensionalen Schemata an der Benutzer-Schnittstelle mit einer speziellen grafischen Notation, die eine Erweiterung des bekannten E/R Ansatzes ist, dargestellt. Diese grafische Darstellung wird intern zur Verarbeitung in eine algebraische Beschreibung des multidimensionalen Schemas transformiert.

Abstract

New application areas for databases like data warehousing and OLAP (Online Analytical Processing) deploy the multidimensional data model in order to describe the application domain. Consequently, OLAP systems are represented by a multidimensional database schema to adequately reflect the application semantics.

FIESTA presents a methodology for the evolution of such multidimensional schemas. Core of the thesis is a schema evolution algebra which comprehends a formal multidimensional data model together with corresponding schema evolution operations. Since OLAP systems are typically implemented as additional layer for relational database systems, the processing of sequences of schema evolution operations in a relational database system is presented. To this end, we formally describe how a multidimensional schema can be mapped to a corresponding relational database schema. In order to fully maintain the multidimensional semantics during this transformation, a corresponding meta schema is introduced as extension of the relational system catalogue. For a consistent processing of evolution operation sequences, a transformation to corresponding relational evolution commands is performed. These relational evolution commands adapt the relational database schema together with the instances and update the contents of the meta schema accordingly.

A prototype for FIESTA has been implemented as part of a graphical design environment for data warehouses. In this environment, multidimensional schemas are presented at the user interface by means of a specialized graphical notation. This notation is an extension of the well-known Entity/Relationship approach. For internal processing the graphical representation is transformed to an algebraic description of the multidimensional schema.

Table of Contents

If an elderly but distinguished scientist says that something is possible he is almost certainly right, but if he says that it is impossible he is very probably wrong.
(Arthur C. Clarke)

1. Introduction

Fact is that today's economy is characterized by a constantly growing competition among enterprises. Only correct strategic decisions made by its managers can keep a business alive. As a consequence, reliable information as base for strategic decisions become an essential production factor. Trends that have been leading to this situation are according to [Kur99] the increasingly complex structures of enterprises through mergers together with the increasing relationships between companies, the introduction of new business processes as well as the redirection of existing business processes to a strongly customer-oriented view, the globalisation of markets, customers and enterprises, and new technologies like the internet/world-wide web or electronic commerce.

Enterprises have collected huge amounts of data in OLTP databases for performing their daily business, but this data is neither integrated nor cleansed and thus not suitable for analytical queries. In order to provide reliable, integrated and up-to-date information that can serve as base for analytical evaluations, data is extracted, transformed, cleansed and integrated to a dedicated data warehouse database. This data warehouse can then be queried by a manager for assistance with his strategic decisions which is commonly known as Online Analytical Processing (OLAP). The warehouse database is typically modeled using a multidimensional view on the data because this corresponds to the manager's understanding of his problem domain. Typically, a manager sees his business as facts (e.g. sales figures, repair facts) that are described in the context of dimensions (e.g. customers, location, time). Dimensions are organized using classifications, i.e. the single dimension elements (e.g. the day 06/30/2000) can be classified according to its month (June) or year (2000). This understanding of the application domain is commonly referred to as a multidimensional schema in the database literature. Despite the multidimensional view on the conceptual layer, OLAP systems are typically implemented using relational database systems because of their proven scalability and reliability [Kim96a], [Kur99], [BG+00].

Since the modeling of such a conceptual multidimensional schema is the central task of the OLAP system design and because of the frequent changes of this schema due to the trends mentioned above, this thesis deals with the modeling process of such a multidimensional schema of a data warehouse and focuses on the efficient processing of schema modifications

that lead to a so-called schema evolution. The complexity of schema evolution in OLAP systems arises basically from the mapping of the semantically rich multidimensional data model to the relational schema on the database system layer. As a consequence of this mapping, a given schema evolution task comprises not only modifications of the relational database schema, but also the adaptation of existing data (commonly denoted as instance adaptation).

FIESTA, the name of our work, stands for "A **F**ramework for Schema Evolut**i**on in Multidim**e**n**s**ional Da**ta**bases".

The introduction given here sketches briefly the design process of multidimensional schemas and derives the overall objective of the thesis. The ideas and visions introduced here will be extended to greater detail in sections 2.1 (role of the conceptual multidimensional data model), 3.1 (motivation for FIESTA), and 3.3 (detailed objectives for our approach). The introduction also presents the relevant state of the art in the areas of schema evolution (in relational and object-oriented database systems), schema modification approaches for data warehousing and OLAP, multidimensional data models, and graphical modeling notations for warehouse design. We conclude this chapter by giving an outline of the thesis.

1.1. The Multidimensional (MD) Schema Design Process

Following traditional database design techniques, a well-defined and purely conceptual database schema of the warehouse database consitutes the necessary starting point for building any warehouse solution and offers additional advantages during later modifications. Surprisingly, this modeling issue receives only little attention both in industry and academia, although existing conceptual modeling techniques cannot be directly applicated due to the peculiarities of the multidimensional data model [SBHD98] (see chapter 2.4). As a consequence, many industrial projects skip the conceptual modeling phase and start either with the logical design (i.e., modeling a relational star or snowflake schema) or – even worse – with the design of a tool-specific database schema. Only recently some approaches have come up in the scientific literature. Yet it is unclear which impact these approaches will have to industrial projects.

In order to fill this gap, we briefly sketch our vision of the ideal data warehouse design process and derive peculiarities for data warehouse schema design and maintenance.

The typical process of the schema design in such an environment is shown in figure 1-1, taken from [SBHD98]. The schema is mainly influenced by user requirements and the availability and structure of the data in operational systems. Most data warehousing projects take an evolutionary approach, both in the warehousing literature ([Kim96a], [Inm96]) and from our experience in several industrial projects (see e.g. [HBD+97]). The projects start with a prototype providing a certain functionality and set of data. This prototype will be further adopted according to the changing and growing requirements gained from users' feedback. Thus, in warehouse maintenance, the user requirements are subject to frequent changes.

In order to assure the flexibility and re-usability of the schema in such an environment, the schema must be specified on a conceptual level. This means especially that it must not assume any facts that are the result of further design steps e.g. the decision which database technology is to be used (multidimensional vs. relational).

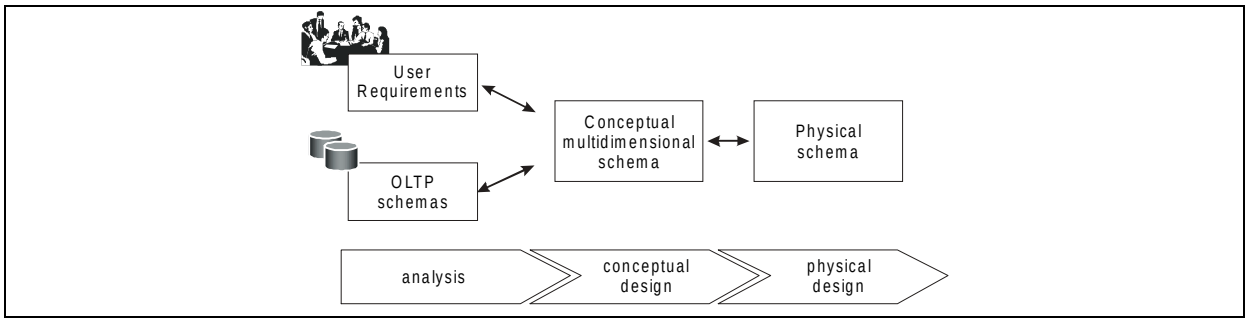


figure 1-1 : schema design process in data warehousing environments

We assume that a conceptual data model reflects the application scenario and deals with no implementation or tool specific details (like the selection of an OLAP tool). For example, an Entity/Relationship diagram is a conceptual model (in the case of relational database design) for us. We will see later why standard E/R models do not reflect the peculiarities of the multidimensional data model adequately. The conceptual model leads to a logical model which corresponds to a database schema. The logical model is implemented through a physical schema which comprises e.g. the disk pages with database tuples and indexes.

As said before, this schema design process has to be performed several times because of the iterative approach that most data warehousing projects pursue. The two main reasons for this very dynamic behavior are:

- the interactive multidimensional analysis technology is new to the knowledge worker. This means that it is impossible for him to state his requirements in advance.
- the business processes in which the analyst is involved are subject to frequent changes. These changes in business processes are reflected in the analysis requirements. New types of queries that require different data become necessary. Since the multidimensional schema of an OLAP system determines the possible analysis capabilities, the new query requirements lead to changes of the MD database schema.

As a consequence, the process of figure 1-1 has to be modified to a more cycle oriented process model which is shown in figure 1-2. This cycle model for warehouse design and maintenance basically consists of the following phases:

- during **'Requirement Analysis',** the requirements of the users concerning data scope, granularity, structure and quality are collected. The result is typically a set of multidimensional views (external schemas) which have to be supported by the information system.
- the main goal of the **'Conceptual Design',** is to consolidate the required views into a single conceptual multidimensional model. During the first iteration a conceptual model is created. During each further iteration of the cycle, the schema developed during the previous iteration has to be modified in order to fulfill the new requirements. The conceptual design is the most important step of the data modeling process as the conceptual schema serves as a basis for the next steps of the cycle and for further iterations.
- during **'Logical and Physical** (Technical) **Design',** implementation decisions are taken. Typical decisions are: which products and architectures to use or which optimization and tuning measures are to take (e.g. denormalisation, precomputation).
- the following **'Implementation'** is a rather mechanic realization of the specifications developed during the technical design phase. Included in this phase is the initial data load (for

the first iteration) or the adaptation of the existing database schema and contents (schema evolution during subsequent iterations).

- during the **'Operation'** phase new data is loaded to the database on a regular basis and the users analyze data. During this phase new requirements for different or differently structured data arise. If a certain amount of new requirements is reached, a new iteration is started.

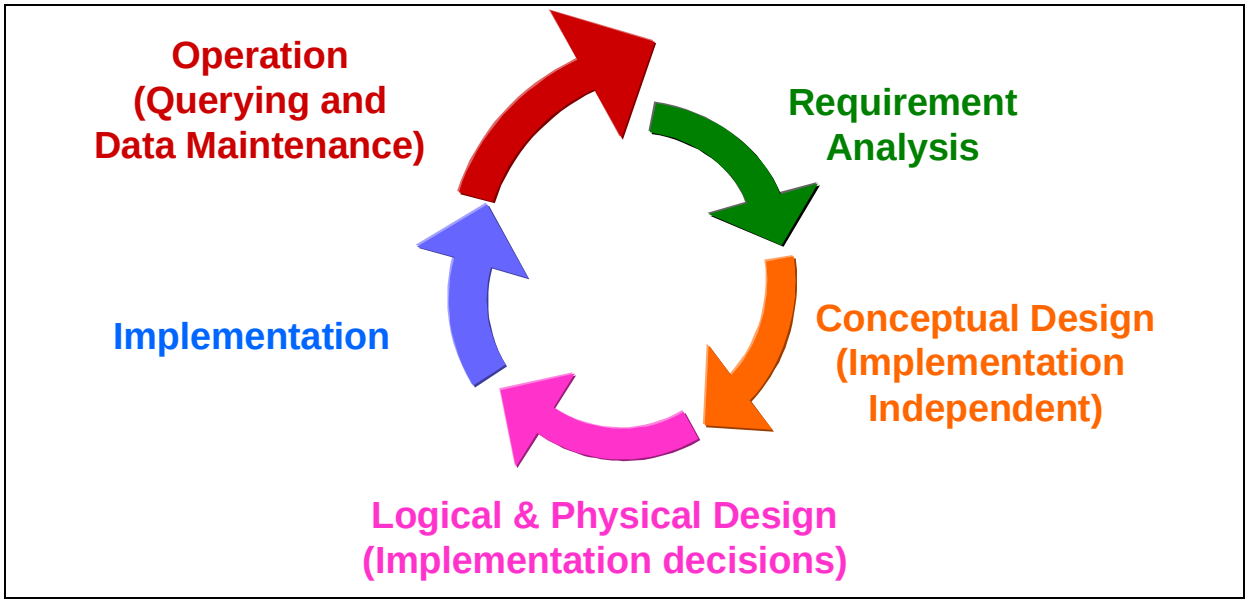


figure 1-2 : enhanced schema design process as design and maintenance cycle

As already mentioned, the conceptual multidimensional data model is the central part of the design and maintenance cycle as it already contains a consolidation of all user requirements (thus describing the business context) but does not yet contain implementation details. All data models that occur later in the design process (e.g. the tool specific database schema) are refinements of the conceptual model.

1.2. Overall Objective of the Thesis

FIESTA aims at a framework supporting schema evolution for OLAP systems.

To that end, FIESTA is embedded in a tool-supported environment which allows the warehouse modeler to specify and maintain his multidimensional schema on a purely conceptual level, thus providing a single point of control to the warehouse modeler. The FIESTA environment automatically propagates the schema modifications (which are expressed by a sequence of schema evolution operations) to the underlying database system. As we will see later, this task comprises modifications of the database schema, an adaptation of existing instances representing OLAP data, and updates of the contents of the FIESTA meta schema.

This high-level vision will be refined to detailed objectives in chapter 3.3.

The following chapter will focus on different research areas that have to be considered for the FIESTA approach. We will elaborate the current state of the art in research and, in our conclusions, show the gaps between this state of the art and our overall vision for FIESTA.

1.3. State of the Art

The objective of this chapter is the presentation of the relevant state of the art for FIESTA. Since FIESTA discusses a research issue (schema evolution) which has been thoroughly investigated in other areas of database research and presents a solution of this issue for an emerging new kind of databases, we present the state of the art in the following areas:

- schema evolution in relational databases (chapter 1.3.1): this is the starting point for all discussions of schema evolution issues. Schema evolution for RDBMS is not only interesting because of historical reasons (i.e. RDBMS have been the first kind of database systems with considerable commercial impact that lasts until today and will still hold for the future), but also because of the poor support of commercial products for schema evolution and instance adaptation.
- schema evolution in object-oriented databases (chapter 1.3.2): in the area of object-oriented database systems, schema evolution became a research topic because the support for schema evolution was a strong request from the application areas of OODBMS. Especially, the complex inheritance hierarchies in object-oriented database schemas were a special case for all approaches dealing with schema evolution issues. As a consequence, numerous publications (some of which lead to commercial products) on this field have been published.
- schema modification approaches for data warehousing and OLAP (chapter 1.3.3): this is the research area in the field of data warehouse research which comes closest to our own approach. However, there are bigger differences that will be pointed out. Especially, we do not follow the viewpoint that reduces a data warehouse to be merely a materialized view. This viewpoint does not adequately reflect the peculiarities and semantics of the multidimensional data model. The versioning approaches are to some extent a generalization of our approach, but we see our approach closer to the user requirements of the specific area of deployment.
- multidimensional data models (chapter 1.3.4): the first step when regarding schema evolution issues from a scientific viewpoint is to fix the meaning and extent of schemas and instances. Thus, when investigating multidimensional schema evolution, the starting point is a formal multidimensional data model. Since there is no commonly accepted multidimensional data model (or, more precisely: formalization of the multidimensional data model), we briefly present the relevant state of the art including our own research results in the larger project context.

For each approach discussed as state of the art, we introduce the main concepts and ideas and try to match the approach with our overall vision for FIESTA, as presented in chapter 1.2.

Finally, we conclude our state of the art presentation with a summary of the main results.

1.3.1. Schema Evolution in Relational Databases

Schema evolution is only poorly supported in relational database systems which constitutes to some degree the root problem for schema evolution in general. Of course, schema evolution in relational databases can always be performed. The DDL and DML commands offer the required expressiveness to adopt a relational table together with its instances. But, due to the

generality of the relational model, there is no means to check consistency for a given application area but this resides in the responsibility of the database administrator.

Modifications of a database schema happen quite often. A study measuring the frequency of schema evolution has been done in the context of a health management system [Sjo93]. The result of this study revealed that the number of relations increased by 139% (during the system's lifetime), the number of attributes by 274%, and that every relation had been modified. Another report [Mar93] concluded that 59% of the attributes are changed in the average case.

The standard SQL DDL allows for changes of a table definition by adding, removing or renaming attributes (columns), setting (or changing) default values, and by modifying constraints (adding/removing primary or foreign key constraints, check constraints). The typical SQL ALTER TABLE command (cited from [Vos94]) is shown in figure 1-3.

```
ALTER TABLE table-name
      { ADD [ COLUMN ] column-name data-type
            | ALTER [ COLUMN ] column-name
                  { SET default-definition | DROP DEFAULT }
            | DROP [ COLUMN ] column-name
            | ADD [ CONSTRAINT constraint-name ]
                  { { PRIMARY KEY | UNIQUE } ( list-of-column-names )
                     | FOREIGN KEY ( list-of-column-names ) REFERENCES ...
                     | CHECK ( condition )
                   }
            | DROP CONSTRAINT constraint-name
      }
```

figure 1-3 : standard SQL ALTER TABLE command

As an example of a commercial system, we refer to the ALTER TABLE of the Informix Dynamic Server [Inf98a]: the ALTER TABLE command allows to add, drop, or modify columns. Modifications extend to a rename, changes of the default-value, constraint definitions and other typical examples. The support for an adaptation of the existing instances basically consists of defining default values for newly added columns or SQL UPDATE queries.

We conclude that schema evolution in relational database systems is somehow trivial (adding or deleting tables and attributes) and can always be done by the database administrator. However, the DBA is responsible for maintaining consistency, especially w.r.t. to the instance adaptation. Typically, this leads to a change specific workaround. We state that the support for the semantics of schema changes offered by today's commercial products is still poor.

Since RDBMS still are typically the backbone of every database application, it is more urgently needed than ever. If relational schema evolution would have been clearly solved, all other directions of schema evolution would be easier because the different application semantics (in our case, the semantics of the MD data model) would only have to be mapped to the existing schema evolution support.

1.3.2. Schema Evolution in Object-Oriented Databases

Schema evolution support became a research issue (with strong impact on commercial products) with the upcoming object-oriented database systems. Object-oriented database systems provide a semantically rich data model, compared to conventional relational database systems.

As a drawback of the object-oriented data model, schema modifications made on one schema entity can have impacts on other schema entities. More precisely, when e.g. the schema of a class is modified, schema modifications of all subclasses have to be performed as non-local changes of the schema. Schema evolution support is an important requirement for object-oriented databases due to the highly dynamic applications (that require frequent schema modifications) for which object-oriented DBS are used.

Similar to the case of OLAP databases, most object-oriented database systems are implemented as an additional layer on top of a relational DBS.

[BKK+87] was the first approach that presented a classification of schema evolution operations for the OODBS ORION (base for the commercial product ITASCA). These operations included e.g. re-naming of classes, adding or deleting attributes, creation or deletion of methods, changes in the inheritance hierarchy, or even defining a new class. As can easily be seen, the type of schema evolution operations depend strongly on the underlying data model (here, the object-oriented data model) [Höf96]. We will come back to this data model dependency of the schema evolution operations in chapter 3.1.

Schema evolution results in OODBS pointed out that schema evolution always comprises the modification of the (database) schema, the adaptation of existing instances, and ensuring a certain well-defined consistency criterion (e.g., a correct association of instances to their classes). Several publications discuss different execution models for the instance adaptation, namely an immediate adaptation, delayed (e.g., at the first writing access), or never. In the last case, specialized filters are constructed (this approach was called screening) in order to ensure correct instances. The contradictory objectives flexibility vs. runtime performance of the overall application favor either one instance adaptation model or another. For the task of schema transformation, most approaches select either a sophisticated view model or version the database schema (or classes, resp.). Another important research issue is the compatibility of application programs during schema evolution (referred to as forward compatibility): how can it be ensured that an application program still works with the modified schema? Here, the sophisticated versioning and view mechanisms find their application.

Schema evolution for object-oriented database systems has been a very busy research area in the beginning 1990's. Numerous results in form of conference and workshop papers, PhD theses (e.g., [Sch93], [Tre95], [Höf96] to name the most popular theses), and research prototypes have been published. Many of the prototypes were further developed to commercial products.

1.3.3. Schema Modification Approaches for Data Warehousing and OLAP

All the research work mentioned above was dealing with schema evolution, but had no direct relationship with data warehousing. Now, we turn to the core research results in the area of data warehousing.

An excellent overview of research problems in data warehousing is given in [Wid95]. Here, support for schema changes during warehouse evolution was requested for the first time. Another overview of research activities is given in [CD97], whereas a more recent summary of advances and open research issues is given in [SMK+98].

The busiest area in data warehouse research is the view maintenance problem. An extended overview of the view maintenance problem (not only in the context of data warehouses) is given in [GM95]. The general idea is to assume the data warehouse database as materialized view over the operational data sources. When data is updated in the data sources, the view has to be updated accordingly. In order to avoid the full recomputation of the materialized view, specialized maintenance techniques have been developed. Incremental view maintenance has become a busy research area, see [GM95] as index to further publications. An interesting case is the issue of the self-maintainability of views. A view is called self-maintainable if it can be maintained using only the materialized view and key constraints [GJM96], [QGM+96]. However, since the view maintenance problem focuses on maintaining the warehouse data during *data changes in the sources* and not on maintaining data during *schema modifications of the warehouse database*, it complements the research contribution of FIESTA, but is not directly relevant as state of the art.

Nevertheless, there are several approaches in materialized view and data warehousing research that are relevant for FIESTA. We briefly introduce the following approaches:

- the view adaptation and synchronization approaches of Bellahsene, Mohania/Dong, and the Rundensteiner research group,
- the temporal approach of Chamoni and Stock,
- the TEMPS approach of Günzel,
- the approach of Hurtado/Mendelzon/Vaisman, and
- the approach of Quix.

1.3.3.1. View Adaptation Approach of Bellahsene

The approach of Bellahsene [Bel98] distinguishes two kinds of schema changes: changes in the operational data sources that lead to changes in the materialized view and direct changes of the materialized view definition.

The paper introduces an extended relational view model which e.g. allows for adding or hiding attributes in the view definition. Hiding is used to simulate deletions of attributes in the view.

Additionally to the view model, operations for schema changes are introduced. The operations concerning schema changes in the operational sources comprise adding and deleting attributes or changing the type of an attribute. Depending on the effects to consistency, these schema changes are reflected in the view definition.

Similarly, the operations for direct changes of the view definition allow for adding or deleting attributes and again for type modifications of an attribute. Changes in the view definition are simulated using the extended view model.

The approach presents a model which is strongly oriented to the semantics of the relational model. Especially, it only focuses on general attributes (of relational tables) and does not treat

the specialized semantics of the multidimensional data model. There is no operation to change e.g. the classification in a hierarchy, or to introduce a new fact.

1.3.3.2. View Adaptation Approach of Mohania and Dong

The view adaptation approach after redefinitions of the view was first introduced in [GMR95] and then enhanced and extended in [MD96] and [Moh97].

The warehouse is assumed to be a materialized (SPJ) view. The problem is how to adapt the view data when changes of the view definitions occur. The key question is of course how to avoid the costly re-computation of the modified view. Thus, adapting the contents of the materialized view seems a promising approach. To that end, [MD96] introduce adaptation algorithms for changes in the `SELECT`, `FROM`, or `WHERE` clause of the view definition. The employed base technique is to add join count attributes to the schemas of the base relations and derive count attributes to the schemas of the views. Sophisticated view maintenance algorithms use this additional information to adapt the view contents without re-computing the view wherever possible.

1.3.3.3. View Synchronization Approach of Rundensteiner/Lee/Nica/Koeller

The database research group of E. Rundensteiner at the Worcester (Massachusetts) Polytechnic Institute has a long lasting tradition in schema evolution research. Starting with lots of publications in the area of object-oriented schema evolution, the group continues their work (see e.g. [CNR99] for a recent publication on object-oriented schema evolution) and transfers existing results to the area of view synchronization during warehouse evolution ([RLN97], [KRH98], [LKN+98], [NR99], [RKZ+99], [LKN+99], [Zha99], [Nic99]).

Numerous publications cover a wide range of different issues, including wrappers for view maintenance [DZR99], general data warehouse maintenance [ZR99] or in the context of schema and data updates [ZR98], parallel view maintenance [ZRD99], and query rewriting [LKN+99].

[RLN97] proposes a taxonomy of view adaptation problems and identifies the view synchronization problem which arises with changes in the source schemas, as a new view adaptation problem. The Evolvable View Environment EVE is introduced as a framework for solving this problem.

Any changes to the view definition or the view extent (i.e., materialized view data) are referred to as view adaptation process in EVE. The proposed taxonomy covers materialized view maintenance [GM95], view redefinition [GMR95], [MD96] (called view adaptation in the original publications), and of course view synchronization.

View synchronization is a dynamic process that adapts the view definition triggered by capability (i.e., schema) changes in the data sources (e.g., the deletion of an attribute). More precisely, the view definition is not changed explicitly (e.g., by the warehouse admin), but by a trigger that was fired due to schema changes in the sources. Additionally, it is assumed that the view extent (i.e., materialized view data) has to be maintained acoording to this view definition change. The latter process is called *view maintenance after synchronization* and can be compared to the instance adaptation phase in the schema evolution literature.

The traditional materialized view problem [GM95] is called *materialized view maintenance after base relation updates* in the taxonomy of [RLN97] and mainly characterized by changes in the source data and no changes in the view definition. The *view maintenance after view redefinition problem* of [MD96] (also renamed from the original approach) is characterized by

explicit changes of the view definition and no (data) changes in the operational sources. This problem comes close to the issue of self-maintainability of views which corresponds to the characterization in the original publication [MD96].

1.3.3.4. Temporal Approach of Chamoni and Stock

The approach of Chamoni and Stock ([CS98], [CS99], also in [BG+00]) aims at modeling temporal multidimensional data in OLAP systems.

Basic idea is to assign a valid time interval (w.r.t. a selected granularity, called *chronon*) to every classification of a dimension. Thus, the complete classification hierarchy (i.e., not only the classification nodes (dimension elements), but also their classification information) is related to the time dimension. When considering e.g. the product dimension as example, not only the insertion or deletion of new products can be modeled, but also re-assigning a product to another product group. Additionally, new classification hierachies can be modeled and represented.

A time stamping technique represents the evolution of data. For every classification relation between two classification nodes, the valid time for this classification is stored, leading to a consolidation tree for the dimension. An equivalent representation are matrices for valid time stamps which have all classification nodes as rows and columns and the time stamps as matrix entries. As drawback, the overall valid time of a classification node cannot be determined because a product may still exist but has not been produced during a certain period or in a certain production plant. Thus, an additional valid time matrix for the consolidation tree has to be provided representing the overall valid time for each classification node of a dimension.

The approach is purely conceptual. So far, no implemented prototype exists, but an implementation on top of a temporal DBS seems promising. Experiences of this implementation could serve not only warehouse research, but would also be useful for the area of temporal DBS as application. Further, performance results in a real OLAP scenario might be interesting.

1.3.3.5. Versioning Approach of Günzel

The latest approach that is relevant as state of the art for FIESTA, is the TEMPS approach of Günzel (published in [Gün00] and in [BG+00]). TEMPS stands for Time-enhanced Multidimensional Processing System and focuses on providing time information for both schema and base data versioning.

The requirements that lead to TEMPS are complex schema changes (e.g., a new product hierarchy every month, yearly new dimensions, changes in the granularity) and changes in the base data (e.g., changes in geographical assignments like villages to counties or variants of analysis). These requirements are motivated from industrial projects with a large market research company. Core of TEMPS is a framework for data warehouses, including versioning aspects (“any data, any time, any analysis”).

To that end, TEMPS defines a temporal multidimensional data model and offers versioning (using time stamps) for classification schemas and hierarchies as well as for the multidimensional cube schema and instances. Specialized evolution operations that describe changes of both the classification schemas and the cube schema are introduced.

1.3.3.6. Dimension Updates of Hurtado/Mendelzon/Vaisman

The approach of Hurtado et al. ([HMV99a] and [HMV99b]) proposes a set of schema evolution operations that are designed specifically for the multidimensional data model (implemented as a materialized view). The multidimensional data model of [CT98] is used.

The authors introduce a formal model of changes in the dimensions (but not the facts) of a multidimensional schema. A distinction is made between changes in the classification hierarchy (schema of the dimension) and changes of the classification nodes (dimension members / instances of the dimension levels). To that end, special operations are introduced that perform the effects of the operations on a materialized view. This materialized view is responsible for the persistent storage of the data.

Further, algorithms are presented to efficiently maintain the materialized view. It is assumed that a fully materialized data cube [GBL+96] is responsible for storage of the view. Especially, all possible aggregates of this data cube have been pre-computed and must be maintained.

1.3.3.7. Warehouse Evolution Approach of Quix

The work of Quix is embedded in the DWQ framework. DWQ is an European research project, dealing with Data Warehouse Quality. The main topics and issues covered by the DWQ project can be found as an in-depth project overview in [JLV+00].

The DWQ framework contains among other a detailed quality meta model and as refinement a quality-oriented data warehouse process model [JJQ+99]. [Qui99] extends this prcoess modell and proposes a framework for data warehouse evolution. The paper regards the creation/update of materialized views, adding/deleting data sources, or changes in the enterprise business model as typical evolution cases that have impacts on the overall quality goals.

In order to control this warehouse evolution, specialized meta data is provided which tracks the history of changes and provides consistency rules to enforce consistency when certain quality factors have to be re-evaluated. To that end, a meta model for the data warehouse evolution as specialization of the data warehouse process model is introduced.

As an example, the evolution of materialized views is discussed. Here, the framework is applied for monitoring data warehouse quality under evolution. Schema evolution operations defined on the relational view model like add base relation/view and their impacts on quality factors are discussed. Mostly, these operations affect the quality factors completeness, correctness, and consistency between the conceptual and logical schema.

The approach has been implemented using the repository system ConceptBase [JGJ+95].

1.3.4. Multidimensional Data Models

A multitude of multidimensional data models (or more precisely: formalizations of the multidimensional data model) has been published in the last few years. Surprisingly enough, there is still no commonly accepted multidimensional data model as this is the case for the relational model.

Surveys and in-depth comparisons of the existing approaches can be found in [BSHD98], [SBH99], [VS99], and [BG+00]. We refer the interested reader to these comprehensive publications for further details.

Here, we do not aim at providing a full in-depth overview, but try to introduce the most prominent models or the models that have been used as base for other approaches. Therefore, some newer approaches like [PJ99] or [DKPW99] are not presented here. We also focus on

the data model and omit the OLAP operations which most approaches also introduce because the operations are not relevant for FIESTA. The considerations and presentations given here may serve as base for the understanding of the FIESTA multidimensional data model in chapter 3.5.

Our idea of the MD data model history and how some models seem (in our opinion) to have influenced other approaches is sketched in figure 1-4. In the beginning, there was the data cube operator of Gray et al. (published in [GBL+96] and [GCB+97]). This approach was closely related to the relational data model and the SQL language. Soon after this approach, the grouping algebra of Li and Wang [LW96], the approach of Agrawal, Gupta and Sarawagi [AGS97], the approach of Gyssens and Lakshmanan [GL97], and the first version of the Cabbibo and Torlone approach [CT97] were published. Whereas the grouping algebra of Li and Wang and the approach of Gyssens and Lakshmanan basically constitute an extension of the relational algebra, the approaches of Agrawal, Gupta, and Sarawagi and Cabbibo and Torlone prefer a pure cube-oriented model.

Thereafter, both refinements of the existing approaches (like [CT98] or [Vas98]) and models providing extended concepts like features [Leh98] or nested cubes [DKPW99] were developed.

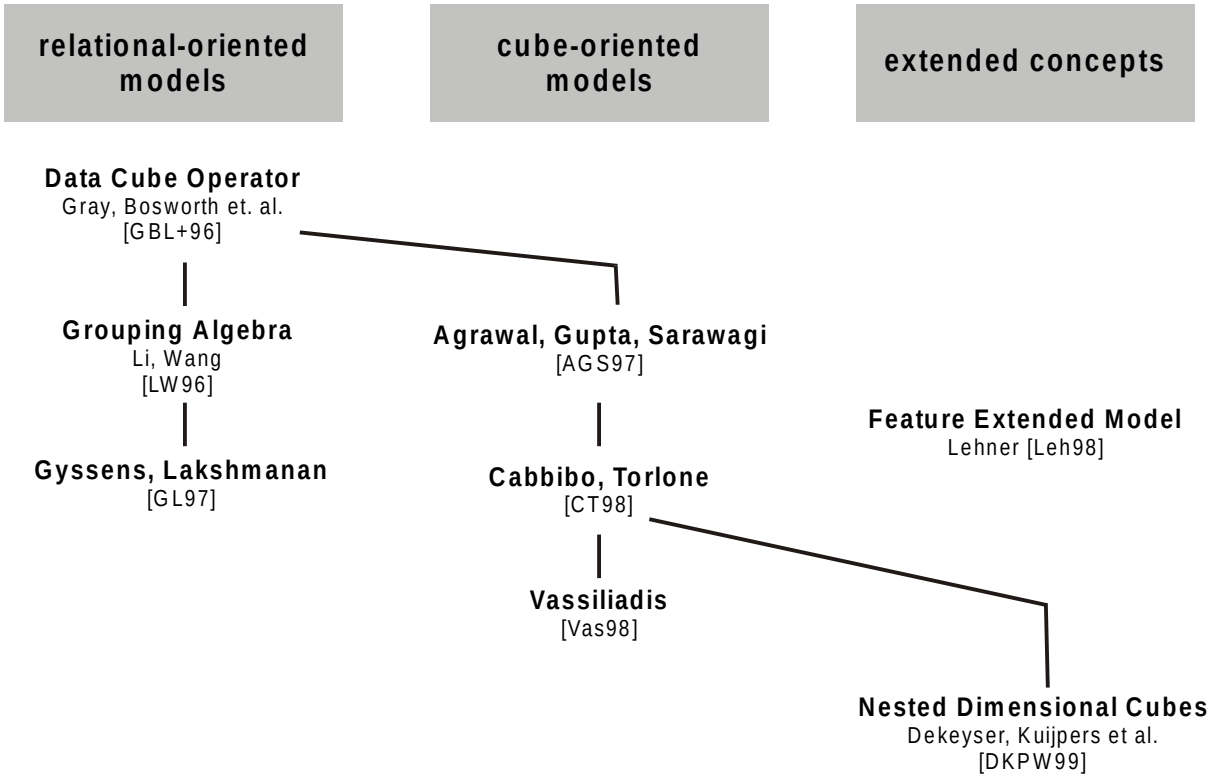


figure 1-4 : overview of MD data model history

In the following paragraphs, we will briefly introduce the multidimensional data models of

- Agrawal, Gupta, and Sarawagi (chapter 1.3.4.1)
- Cabbibo and Torlone (chapter 1.3.4.2)
- Li and Wang (chapter 1.3.4.3)

- Gyssens and Lakshmanan (chapter 1.3.4.4)
- Lehner, Ruf, and Teschke (chapter 1.3.4.5) and
- Vassiliadis (chapter 1.3.4.6)

Due to the strongly relation-oriented view, we do not introduce the approach of Gray et al. ([GBL+96], [GCB+97]). As already mentioned, we do also not introduce some newer approaches for rather specific requirements like [PJ99] or [DKPW99]. These approaches do not support our reader in understanding the FIESTA multidimensional data model.

For the reader who is rather new to the area of multidimensional data models, we recommend reading one of the overviews ([BSHD98], [SBH99]) or chapter 2 of this thesis first in order to understand the terms and requirements.

1.3.4.1. Model of Agrawal/Gupta/Sarawagi

In [AGS97] a pragmatic multidimensional data model and an algebra are presented. It organizes data in one or more hypercubes. All cell values can either be an n-tuple or from the set {0,1}. A cell containing "1" means that this combination of dimension values exists. An n-tuple represents the existence of a record with n measures and a "0" marks cells with no contents. The dimensions have no structure or order and the elements are addressed by their name.

A k-dimensional cube C with n-tuples as cell values is formally defined as a triple (D, E(C), N) where D is a set of k dimension names. Each dimension has a domain dom_i. E(C) is a function mapping dom_1 x...x dom_k to an n-tuple (the cell values of the cube C) or to {0,1}. N is an n-tuple containing the names of the members of the n-tuples contained in the cube.

The approach does not explicitly distinguish between structure and contents. The model does not contain any information about the structure of the dimensions. Especially, there is no static construct representing dimension levels. That means all of the structural and functional information has to be included in the query. For this purpose, the *merge* operation allows to supply a dimension merging function (for the structural mapping) and an element combining function (for functional definition).

As n-tuples are allowed as cube elements, record structured measures can be expressed easily. Derived measures can be expressed by using a self-join operation on the cube. In this case the definition of the calculation has to be given in the query. The expressive power of the model is at least as powerful as the relational algebra as the relational operators projection, union, intersect and difference can be expressed using the basic operator set.

1.3.4.2. Model of Cabbibo and Torlone

L. Cabbibo and R. Torlone proposed a formal multidimensional model and a corresponding descriptive query language based on a logical calculus ([CT97], [CT98]). The multidimensional data model is defined by the notion of f-tables as basic data structure. F-tables are relations that contain a tuple for each cell of the data cube containing a value. Dimensions are defined by graph (DAG)-structure containing dimension levels as nodes.

Formally a dimension is defined as a triple (L, $\leq$, R-UP). L is the finite set of levels which is partially ordered by the relation $\leq$ (e.g. *garage* $\leq$ *region* means level *garage* rolls up to level *region*) . R-UP is a collection of roll-up functions that define the mapping of lower level elements to higher level elements (e.g. *garage A, B* and *C* belong to region *bavaria*). Each level

$l \in L$ is associated with a countable set of values called the domain of L (e.g. dom(garage) = {A,B,C...}).

N-dimensional f-tables, the central modeling entity have the following form: $f[A_1:l_1,...,A_n:l_n]:l_0$. f is the name of the f-table, l_i ($0 \leq i \leq n$) is the name of a dimension level and A_j ($1 \leq j \leq n$) is the name of an attribute.

A multidimensional scheme is defined as a tuple (D,F) where D is a finite set of dimensions and F is a finite set of f-tables over these dimensions.

The treatment of multiple hierarchies on one dimension is easily possible as the relation $\leq$ defines only a partial order on dimension levels. The formalism does not introduce explicit names for the different possible aggregation paths.

Complex measures can be treated in two different ways. First a single f-table can be defined for each measure. This does not allow derived measures. Another possibility is to define the measures as an own dimension. With this solution it is possible to express derived measures by using dimension levels with the disadvantage that the functional definition of the levels has to be included in every query accessing the derived measure. But then all atomic measures must be of the same domain (e.g. numeric).

In [CT98] an extension of the approach is published. It extends the MD model to support record structured measures and defines an algebraic and a graphical query language. The algebra makes use of ten operators many of which are similar to relational operators (e.g. join, cartesian product, selection etc.)

1.3.4.3. Model of Li and Wang

The work of Li and Wang [LW96] formalizes a multidimensional data (MDD) model for OLAP applications. Core of the approach is an algebraic query language, called grouping algebra. Basic concept is a multidimensional cube consisting of a number of relations, the dimensions, and for each combination of dimension tuples, an associated (scalar) data value representing a single fact attribute. The paper introduces an MD cube algebra for manipulating such cubes.

An n-dimensional cube scheme is a set $\{(D_1,R_1),..., (D_n,R_n)\}$ with D_i being the dimension names and R_i being sets of attribute names. An MD cube on such a scheme is a pair (F, μ) where $F=\{(D_1,r_1), ..., (D_n,r_n)\}$ with r_i being a relation on R_i for each i and μ is a mapping from $\{\{(D_1,t_1), ... ,(D_n,t_n)\}| \forall\ 1 \leq i \leq n: t_i \in r_i\}$ to V (set of scalar values). Informally, a cube is a set of dimension relations r_i and a mapping from an n-dimensional tuple (coordinate) to a scalar value. The paper introduces a grouping algebra on MD cubes with relational operations (rename), order-oriented operations (roll) and an aggregation operator.

A multidimensional database is a finite set of MD cubes and a finite set of grouping relations. The MD cube algebra serves as query language. Operations of the MD cube algebra are *add dimension*, *transfer*, *union* of cubes, *cube aggregation*, *rc-join* (join a relation into a dimension of a cube), and *construct* a cube from a relation. One of the main features of the approach is the fact that it includes 'regular relations' so that the algebra can be seen as an extension of the relational algebra.

The grouping algebra provides an implementation independent, declarative approach to multidimensional analysis and OLAP applications. Queries can be specified straightforwardly as can be seen in the examples given below. Since currently only a mapping to a scalar value is allowed, it is not possible to express complex measures. A possible solution is to build a separate cube for each measure attribute. Derived measures have to be computed separately.

Dimension hierarchies and multiple hierarchies can be expressed by using the very powerful grouping mechanism and the corresponding operators (roll, order, aggregation). The expressiveness of the proposed operators is strong and specifically designed for typical OLAP applications including e.g. TopN or cumulative sums.

As a shortcoming of the approach we remark that as fact attribute there is only a single scalar value allowed. Therefore, for facts with a set of measures, a separate cube has to be defined for every single measure.

Summarized, we may say that the model of Li and Wang is a real conceptual model, has a strong expressiveness through the powerful operators, but needs an extension to provide more complex fact values instead of just a single scalar value. This would allow to model complex facts in one cube instead of multiple cubes based on the same dimensions.

1.3.4.4. Model of Gyssens and Lakshmanan

Gyssens and Lakshmanan introduce a conceptual multidimensional data model for OLAP applications in [GL97]. The authors see the main benefit of their work in a clear separation between structural aspects and contents. They propose an algebra and an equivalent calculus for their model. There are no implementation issues mentioned because the main focus is on the conceptual part.

The basic formalism is as follows: Let *N* be a set of names, *V* be a set of values.

An n-dimensional table schema is a triple <D,R,par> where

$D=\{d_1, \ldots, d_n\}$ is a set of dimension names,

$R=\{A_1, \ldots, A_m\}$ is a set of attributes, and

par: $D \rightarrow 2^{\{A1, \ldots, Am\}}$, such that for all i,j=1,...,n, i≠j, $par(d_i) \cap par(d_j)=\emptyset$, and $\bigcup_{d \in D} par(d) \subseteq R$.

$par(d_i)$ is denoted by X_i. Let $M = R - \bigcup_{1 \le i \le n} X_i$.

An instance of an n-dimensional table schema ⟨D,R,par⟩ is a set of n+1 finite relations of the form $rd_1(Tid,X_1),\ldots,rd_n(Tid,X_n)$, $rm(rd_1.Tid,\ldots, rd_n.Tid,M)$ such that

- the join $\pi_{Tid}(rd_1) \times \ldots \times \pi_{Tid}(rd_n)$ equals $\pi_{rd1.Tid, \ldots, rdn.Tid}$ (rm), i.e., for every combination of Tid values in the relations $rd_1, \ldots, rd_n$, there is at least one corresponding tuple in rm, and every tuple in rm corresponds to some combination of Tid values in the relations $rd_1, \ldots, rd_n$;
- for all i=1, ... ,n, Tid is a key of the relation rd_i ; and
- for all i,j = 1, ..., n, i≠j, $\pi_{Tid}(rd_i) \cap \pi_{Tid}(rd_j) = \emptyset$, i.e., the Tid values in different relations rd_i and rd_j are disjoint.

A multidimensional tabular database (MDD) is a set of tables. This definition of an MD table schema reminds of the common star schema, however, defined on a conceptual level: *par* assigns the dimension attributes to a dimension table, the rd_i are the dimension tables, *rm* is the fact table, the condition *(i)* models a foreign key relation conceptually, condition *(ii)* assures the key property for the dimension tables, and *(iii)* ensures unique Tid values or keys. The approach distinguishes between parameters (i.e., dimension attributes X_i) and measure attributes (elements of M in *rm*).

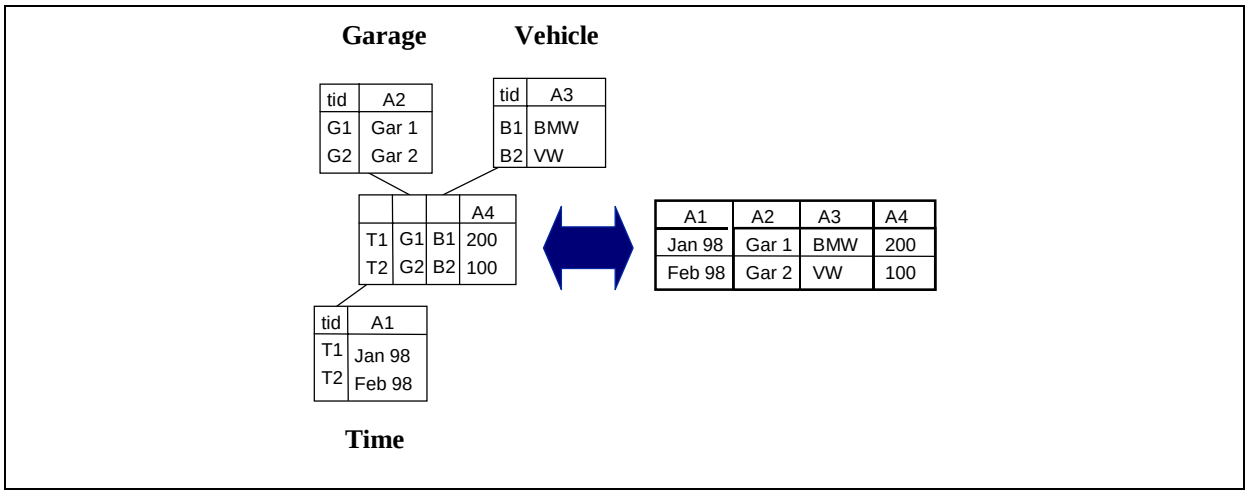


figure 1-5: tabular (left side) vs. relational representation (right side)

It is shown [GL97] that every MDD table can be represented by a classical relation and vice-versa: an example of an MDD table corresponds to the set of tables on the left side in figure 1-5. This tabular representation is shown to be equivalent to the single (classical) denormalized relation (on the right side in figure 1-5 for the example) where the full join (on the Tid values) over all tables has been computed. Consequently, this equivalence allows to base the approach on the relational algebra.

Derived measures must be materialized and stored in the fact table *rm*. Dimension hierarchies are modeled in the attribute names, i.e., every hierarchy is stored as separate attribute of a dimension. Same applies to multiple hierachies on a dimension. Complex measures can be modeled as additional attributes in the fact table rm.

As it can be seen, the simplicity of the approach guarantees simple definitions of the operators as well as simple definitions of the database schema. Specifying queries makes things a bit more complicated, because the tabular algebra provides only the basic constructs (no join operators, no predefined aggregation functions). Anyway, since [GL97] incorporates all first-order definable classification and aggregation functions, all these constructs can be expressed in the approach.

1.3.4.5. Model of Lehner/Ruf/Teschke

[LRT96] contains an extension of the multidimensional model by providing two orthogonal structuring mechanisms for dimensions: classification hierarchies and features. In [BL97] a query language (called CQL) for this enhanced multidimensional data model is presented. [Leh98] contains a formal description of the nested multidimensional data model which supports this extension and an algebra for data manipulation.

Let us first take a look at the structue of a single dimension. According to this approach a dimensional structure (e.g. vehicle) contains a finite number of dimensional elements (or basic objects). Each dimension is characterized by a primary attribute (e.g. Vehicle Id_Nr). The dimensional elements are instances of this attribute (e.g. 10123). Furthermore, a dimension has a list of classification attributes (e.g. vehicle type and brand). Each of these attributes represents a level in the dimensional hierarchy. Instances of these classification attributes represent nodes of the classification tree.

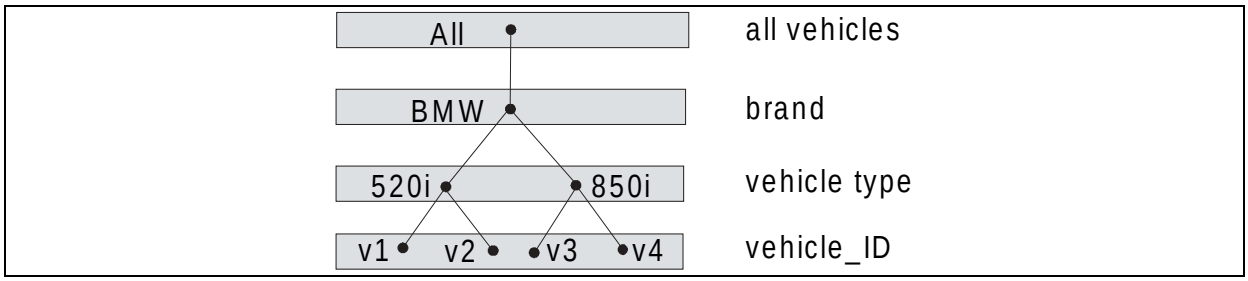


figure 1-6: The instances of classification attributes span the classification tree

Each node (not level) of the classification tree is characterized by a set of dimensional attributes (or features). Notably, the structure of a dimension (containing classification nodes and dimension attributes) is only described informally in the paper.

The approach distinguishes primary and secondary multidimensional objects (MO). A primary MO is formally defined as a quintuple (M, DS, D, t_A, t_D), where

- M is a unique cell identifier and contains the name of the measure
- DS is an n-tuple which contains the dimensions of the cube and their granularity. Each element of the n-tuple is either a primary attribute or a categorization attribute (DS is called context descriptor schema).
- D is a context descriptor (the instance of a context descriptor schema) specifying the selection criteria. This context descriptor corresponds to the WHERE clause in SQL.
- t_A is an aggregation type (sum, avg or none)
- t_D is the data type of the numeric measure

A primary multidimensional object represents the structure of a data cube of a certain granularity according to the classification hierarchies. Thus, it corresponds to the ‘classical’ multidimensional cube definition as it is used by the other approaches. The paper does not formalize the contents of the cube.

Secondary multidimensional objects are used to formalize the extension of the classical multidimensional model by features. Formally, such an object is defined as follows:

- D is a context descriptor which identifies a node of the classification tree and
- DA is a set of dimensional attributes that are applicable to the context descriptor (node of the classification tree)

A multidimensional object corresponding to a cube (containing classification hierarchies and features) is defined by a primary multidimensional object P and a set of dimensional attributes for defining the corresponding nested secondary multidimensional objects.

Compared to other approaches this model seems rather complex. This is partly due to the fact that an extended multidimensional model is formalized.

Only single numerical values are allowed as cell entries, which does not allow the natural modeling of record structured cell values. A distinctive feature of the model is the inclusion of information about the additivity of a measure. Thus, additional application semantics are captured by the model. Albeit, the additivity can only be modeled for a whole cube. Real world applications often require to distinguish additivity along different dimensions (e.g. the sum of “parts in stock” is meaningful along the garage dimension but not meaningful along the time dimension)

As dimension levels are linearly ordered (which is a consequence of modeling levels as a list of attributes), only balanced tree-structured hierarchies can be modeled. As the dimensional structures are only described informally, it remains open if multiple classification hierarchies per dimension are allowed.

1.3.4.6. Model of Vassiliadis

The intention of the Vassiliadis approach [Vas98] is to provide a model which contains the natural OLAP operations (like slicing and drilling) as operators. The author introduces a formal data model and an algebra which can be mapped to the relational algebra as well as to native array data structures.

The basic formal definition of dimensions is very similar to the MD model of [CT97]. A dimension is defined as a lattice (H, ≤). H = {DL_1,..,DL_m}is a set of levels with a domain dom(DL_i) attached to each level DL_1. A distinctive feature is the use of multivalued dimensions (i.e., dimensions that contain members more than once). This allows for elegant operator definitions. The relation ≤ defines a partial order on the dimension levels. Additionally, the notion of dimension path is introduced as a linear, totally ordered subset of the level set. Each dimension contains a set of dimension paths (with only one element if no multiple hierarchies are defined on the dimension). The dimension levels of different dimensions have to be disjoint.

The mapping between dimension members of different levels that belong to the same dimension is defined by the two functions ancestor and descendants.

The multidimensional data is contained in cubes. The approach distinguishes between cubes and basic cubes. A basic cube C_b is formally defined as a triple <D_b,L_b,R_b>. D_b is a list of the dimensions characterizing the cube and also contains a special measure dimension M. L_b lists the atomic dimension levels for each dimension. R_b is a set of cell data containing the tuples of the data cube.

From this base cube further cubes can be derived by a set of operations. Thus, a cube formally is a 4-tuple <D,L,C_b,R>, where C_b is the base cube from which the cube was computed. A multidimensional database is defined as a tuple <D, C> with D being a set of dimensions and C representing a basic cube.

The paper does not give a clear separation of structure and contents as the definition of (basic) cubes contains the dimension structure as well as the data tuples. Furthermore, no user-defined aggregate functions are allowed in the model. By only allowing one multidimensional cube per database, multi-cube models that contain several cubes sharing dimensions cannot be expressed.

According to the authors, a distinctive feature of their work is the inclusion of an explicit drill-down operation into their model. While it is correct that such an operator is not expressed in most of the models, an equivalent operation has also been introduced in [Leh98]

1.3.5. Summary

After having presented the relevant state of the art in several rather different areas, we want to elaborate where the introduced approaches fail in fulfilling the overall objective of our approach. To that end, we have listed criteria of the FIESTA objective (see chapter 1.2) and evaluated the related approaches together with FIESTA in a table (see figure 1-7).

The criterion “purely conceptual approach” refers to the question whether the approach is defined on a purely conceptual layer and does not assume any implementation decisions like the use of an RDBS. The criterion “complete schema evolution algebra” refers to the question if a formal algebra for schema evolution is provided, i.e. a formal data model with schema evolution operations defined on that data model. The next criterion “automatic adaptation of schema and instances” shows if the system environment automatically adapts the database schema and existing data or of this is left to the user / admin. The criterion “tool-supported environment” corresponds to our vision of the “single point of control”, i.e. an environment which allows for schema design and maintenance on the conceptual layer, using a graphical modeling tool.

The different multidimensional data models (chapter 1.3.4) are not contained in the table because none of them deals with schema evolution in general. The presentation of these models shall only assist in understanding the FIESTA multidimensional data model (as part of the FIESTA schema evolution algebra, see chapter 3.5).

The “approach” schema evolution support in RDBMS may seem a little strange here, but since the relational model offers basic schema evolution support and because many approaches are implemented as additional layer on top of an RDBS (and then use the schema evolution support of the RDBS), it must be contained in the table.

	purely conceptual approach	**based on MD data model**	**complete schema evolution algebra**	**automatic adaptation of schema and instances**	**tool-supported environment**
schema evolution support in RDBMS	no	no	yes (part of the SQL DDL)	transformation of schema, manual adaptation of instances	yes
schema evolution approaches in OODBMS	yes	no	yes	yes, but degree of support varies in approaches	yes (some approaches)
approach of Bellahsene	no	no	no, changes concerning attributes in view definitions	yes	no
approach of Mohania / Dong	no	no	no, changes in view definitions	yes	no
approach of Rundensteiner / Lee / Nica	no	no	process-oriented model within EVE	yes	yes, EVE environment

	purely conceptual approach	**based on MD data model**	**complete schema evolution algebra**	**automatic adaptation of schema and instances**	**tool-supported environment**
approach of Chamoni / Stock	yes	yes	only schema changes in dimensions, no formal algebra	no, versioning approach representing classification evolution over time	not implemented
approach of Günzel	yes	yes, temporal MD data model	yes	versioning approach	yet unknown, possibly integration in CubeStar environment
approach of Hurtado / Mendelzon / Vaisman	yes	yes	yes, but algebra not complete	yes, materialized view maintenance	unknown
approach of Quix	yes	no	no, process-oriented model	not directly (by separate view maintenance), effects on quality factors are regarded	yes, DWQ environment and prototype
FIESTA	yes	yes	yes	yes, automatic schema transformation for RDBS, instance adaptation, meta data update	yes, BabelFish tool environment

figure 1-7: FIESTA vision opposed to state of the art

Basic observation is that only the approaches based on a multidimensional data model can and must be compared with the FIESTA approach. As pointed out before (and also shown in chapter 3.1), schema evolution is always strongly specific for a data model.

Thus, the versioning approaches of Chamoni / Stock and Günzel and the approach of Hurtado et al. must be compared with the FIESTA solution. We refer to chapter 5.2 where we discuss what the approaches have in common and where they differ.

1.4. Outline of the Thesis

Chapter 2 provides necessary prerequisites for the understanding of FIESTA. First, the overall vision of FIESTA within the research project BabelFish is presented. Then, we introduce the BabelFish layer model and define basic terminology of the multidimensional data model. Finally, the ME/R modeling technique for multidimensional schema design is presented.

Chapter 3 presents the first part of the FIESTA core, the conceptual schema evolution methodology. Starting with a motivation for FIESTA and a generic roadmap to schema evolution, detailed objectives for FIESTA and a formal approach to multidimensional schema evolution are derived. Next, the schema evolution algebra, consisting of the multidimensional data model and the schema evolution operations are introduced. Since MD schemas are visualized by ME/R graphs in the graphical schema design and maintenance environment, the dualism between MD schemas and ME/R graphs is formally introduced. The chapter concludes with considerations on the processing of schema evolution jobs, corresponding to a sequence of schema evolution operations.

Chapter 4 as second core part describes how MD schema evolution operations can be processed in a relational DBS. To that end, a formal mapping between MD schemas and relational schemas is defined. In order to maintain the multidimensional semantics, the FIESTA meta schema is designed. Next, the mapping is refined to a formal consistency criterion between the conceptual multidimensional layer and the logical database layer. Thereafter we present the transformation of conceptual schema evolution operations to corresponding logical evolution operations. These logical evolution operations transform the relational database schema, adapt the existing data, and update the contents of the meta schema accordingly. The transformation algorithm with its main design decisions and concepts is explained in detail.

Chapter 5 discusses the FIESTA solution. To that end, the BabelFish environment including the FIESTA implementation is introduced. We discuss related approaches and show why and how FIESTA fulfills our overall objective and vision.

Chapter 6 concludes the thesis with a summary of the main contributions of FIESTA and an outlook on future work.

Get your facts first, and then you can distort them as much as you please.
(Mark Twain)

2. Basic Concepts of Multidimensional Data Modeling

In this section, necessary prerequisites for the understanding of the thesis are provided. To that end, we first sketch the overall vision of the project BabelFish into which this thesis is embedded. We informally explain the general idea of a conceptual multidimensional data model and set its focus and role for FIESTA. Next, we introduce the BabelFish layer model that is used throughout the thesis and, consequently, serves as base for the understanding of FIESTA. Since there is no common notion and terminology for multidimensional data models in the literature, we provide the basic terminology for our multidimensional data model (which is formally introduced in chapter 3.5). Finally, we present the ME/R modeling technique (which has been developed in the BabelFish project) and show how it is used for the design and maintenance of conceptual multidimensional schemas within FIESTA.

2.1. Overall Vision and Role of the MD Data Model

tbd Forwiss-Report BF [BSH00]

As already mentioned, FIESTA (which is subject of this thesis) is embedded in the research project BabelFish at the Knowledge Bases Group of FORWISS. Consequently, we start with an introduction of the overall objective of BabelFish and derive the scope and contribution of FIESTA. Further detailed information about BabelFish and its results is reported in [BSH00].

The overall objective of the BabelFish project is to provide a methodology and environment for the tool-based design and maintenance of repository-driven data OLAP systems. Our underlying vision is a tool-based environment where all necessary knowledge is visualized with graphical models and stored in a repository. This environment enables a single point of control for the design and maintenance of the OLAP system.To this end, different modeling and design methodologies have been investigated with respect to their transferability in the warehousing area [SBH99]. In order to capture all aspects of the OLAP design, we distinguish between two orthogonal areas, namely

- static models: for the structure (schema) of the multidimensional cube
- dynamic models: to specify user behavior

When opposing the aspects design and maintenance with the orthogonal views static vs. dynamic, the following table gives an overview of the research activities under the umbrella of BabelFish:

	OLAP design	**OLAP maintenance**
static aspects	ME/R	FIESTA
dynamic aspects	PROMISE	-

figure 2-1 : overview of the BabelFish activities

BabelFish started with static modeling techniques for the initial design of an OLAP system. Since none of the examined methodologies fully reflected the peculiarities of the multidimensional data model (for details, see [BSHD98] and [SBH99]), a specialized graphical notation for the conceptual design of OLAP systems has been developed: the ME/R notation [SBHD98], an extension of the well-known Entity/Relationship modeling technique [Che76]. We describe this modeling technique later in more detail.

Concerning the dynamic aspect of OLAP design, the PROMISE approach [Sap99], [Sap00] investigates how user behavior can be specified (i.e., modeled) and deployed for the optimization of query processing (e.g. by pre-computing the next probable user queries or parts of these queries).

Picking up the idea of the single point of control, BabelFish investigates how conceptual models can be used for automatically generating and maintaining OLAP systems. The objective is the overall maintenance of the OLAP system by the automated analysis and specialized components for generating corresponding programs or parameters. This means that an OLAP designer does not have to know all implementation decisions (e.g. which database system has been used for the implementation, how are dimension hierarchies modeled in the database schema), but the BabelFish tool environment manages the necessary knowledge about implementation decisions and details. This knowledge is stored as metadata in a repository system [BD94].

FIESTA comes into play when the conceptual model is maintained and changed during the lifetime of a running OLAP system [Bla99]. FIESTA formally defines what types of modifications occur on a multidimensional data model (which is visualized by an ME/R model) and provides a tool that automatically propagates these modifications (which are specified using a schema design tool) to the underlying implementation in the warehouse database.

Summarizing, the conceptual multidimensional schema and its modifications during lifetime constitute the single point of control for schema maintenance and thus form the starting point for the approach of FIESTA.

The core idea to manage the whole system using a conceptual design tool allows to derive our layer model accordingly in the next chapter.

2.2. Layer Model

As already mentioned, one of the main research areas and contributions of BabelFish is the conceptual design of OLAP systems with ME/R models.

We assume that an OLAP modeler performs the conceptual design tasks (i.e., multidimensional schema design for the focus of FIESTA) with a graphical modeling tool. Internally, the resulting ME/R diagrams are described by means of a multidimensional algebra. We call this layer the conceptual OLAP layer. An overview of our BabelFish layer model can be seen in figure 2-2.

Layer	Maintained by	Tool / Component	Formalism
Conceptual OLAP layer	OLAP Modeler	Modeling Tool	Graphical (ME/R), MD Algebra
Logical OLAP layer	OLAP Administrator	RDBMS / MDDBMS	Relations / Multidimensional Arrays
Physical OLAP layer	Database Administrator	RDBMS / MDDBMS	System specific (e.g. relational tables with indexes / MD tiles)

figure 2-2 : the BabelFish layer model

The persistent storage of the multidimensional OLAP data is typically performed by either a relational DBMS or a purely multidimensional DBMS. Consequently, the conceptual multidimensional schema is implemented either in a relational DBS or a multidimensional DBS. We assign the corresponding logical database schema to the logical OLAP layer. Typically, this logical schema is maintained by an OLAP administrator. Technically, it consists either of a set of relational tables (in the structure of a so-called star or snowflake schema) or a set of multidimensional arrays/cube definitions.

The end-users typically use OLAP tools as frontend applications. These OLAP tools store their metadata (which e.g. represent the missing multidimensional semantics in case of a relational structure of the warehouse database) also in the warehouse database. Consequently, we define this metadata to be part of the logical (database) schema.

The physical layer is then the corresponding internal layer of the used DBMS. In case of a relational DBMS, this extends to clustering strategies or index design, in case of an MDDBMS to e.g. tiling strategies or sparsity handling. Informally, the physical OLAP layer is concerned with the DBMS-internal storage and management of the data and outside the scope of this thesis.

tbd: noch was zur Abbildung between conceptual and logical layer

2.3. MD Data Model: Basic Terminology

As shown in chapter 1.3.4, where we sketched several multidimensional data models (or more precisely: variant formalizations of the multidimensional data model) that have been proposed and discussed in the research community, there is no commonly accepted formalization and terminology of the multidimensional data model.

In general, a data model provides means to define schemas together with specific operations working on the instances (which are data satisfying the schema constraints). For the multidi-

mensional data model, there is no consensus both of the extent of this means and the terminology.

This section overcomes the shortcoming of existing approaches formalizing the MD data model (although we know that a definition at this point in the thesis is a chicken or egg problem, because for the description of existing approaches we informally used the terminology without having it introduced before) and provides a clear definition of the single parts of a multidimensional data model as well as a terminology which is used throughout this thesis.

The multidimensional data model (like the relational data model) basically consists of a means to define multidimensional schemas, a set of integrity constraints (which can be expressed by means of the schema), and specific operations on the instances (e.g. slice, dice, pivot). For a given schema, instances can be defined as data which is organized according to the schema (or more formally, satisfies the schema constraints). A multidimensional schema together with the multidimensional instances form a multidimensional database (analogously to the definition of a relational database). In order to complete this enumeration from the schema evolution point of view, we add specific operations on the schema (e.g. to change the structure of dimensions or to add measures to a cube). In general, these operations are called schema evolution operations.

The peculiarities of the multidimensional data model arise from the division of the schema (and consequently also the instances) in a dimensional part (often informally defined as qualifying data) which describes the hierarchically ordered dimensions and the multidimensional measures (or quantifying data) which describes the measured data organized in the multidimensional space defined by the dimensions. This is usually visualized using the cube metaphor (see figure 2-3).

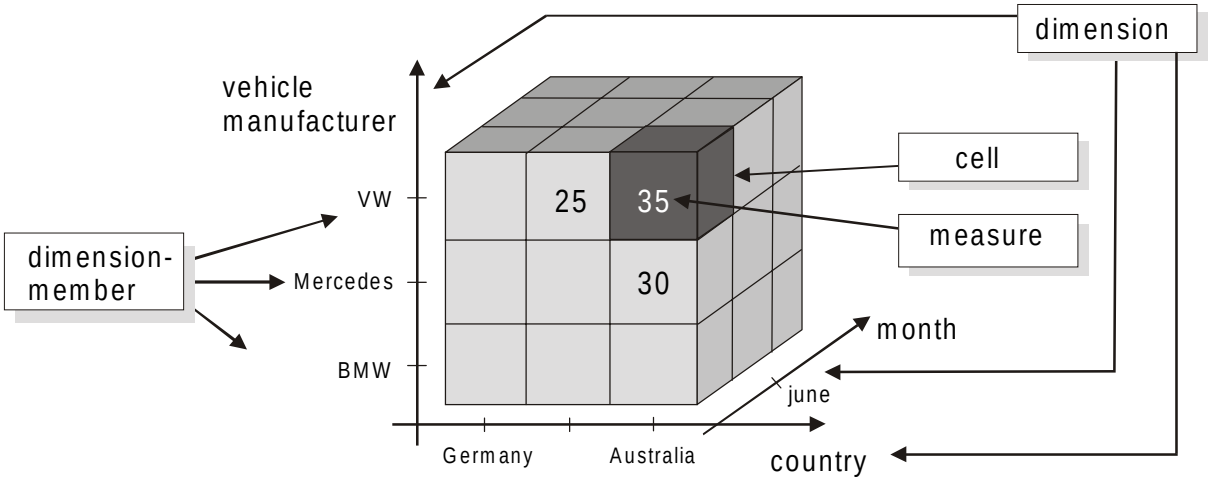


figure 2-3 : the cube metaphor

In this example cube for vehicle repairs, the three dimensions country, month and vehicle manufacturer span the multidimensional space. The dimension members (e.g. Mercedes) define the coordinates of a cube cell. The highlighted cell has the coordinate (Australia, May, VW). The measure value of this cell (e.g. 35 vehicle repairs) is depicted in the highlighted cell. A cell can contain not only a single measure, but a set of measures, e.g. the number of repairs together with the total part costs and total personnel costs.

Dimensions are hierarchically organized according to classification relationships. Graphically, a dimension schema can be visualized by a directed, acyclic graph (DAG). A sample dimension schema is shown in figure 2-4. The boxes represent the dimension levels and the arrows represent the classification relationships between the dimension levels.

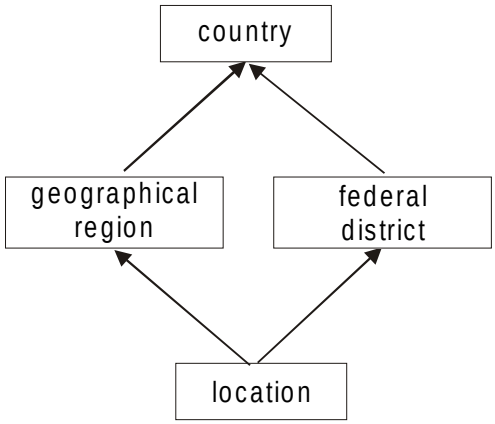


figure 2-4 : sample dimension schema

In addition to the dimension schema, the classification hierarchy for the instances of the corresponding dimension levels has to be defined as well. A sample classification hierarchy is depicted in figure 2-5. Each *day* (represented by a classification node) is assigned (visualized by the edges) to a *month* which represents its classification according to the higher dimension level *month*. Note that this hierarchy is not only used for aggregation (e.g. monthly car sales figures), but also for navigation purposes.

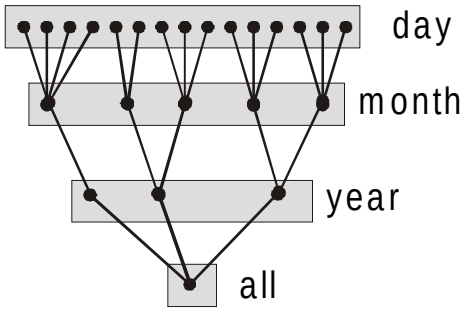


figure 2-5 : sample classification hierarchy

The dimensional schemas for all dimensions together with all classification hierarchies (which correspond to the dimensional instances) form a dimension.

The interesting issue in the case of the multidimensional data model is the dualism of the classification hierarchies. On one hand, they constitute the instances for the dimensions and on the other hand, they are part of the multidimensional cube schema. The latter option is visualized in the following figure 2-6:

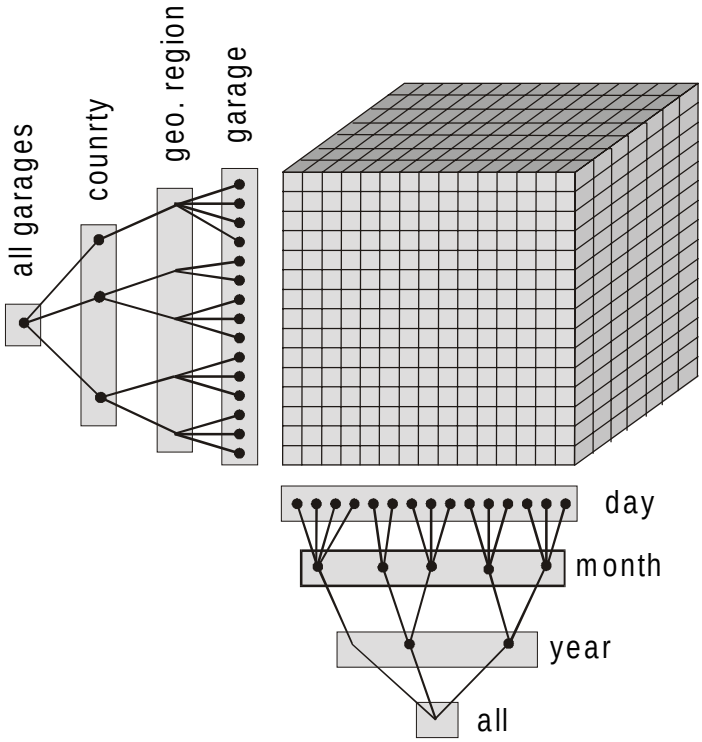


figure 2-6 : multidimensional cube with classification hierarchy

This visualization shows that depending on the selected granularity of the multidimensional measures, the dimensional classification nodes define the schema (i.e., the coordinates of the multidimensional space) of the multidimensional measures (represented by the cube cells). Consequently, the schema of the single cells of the multidimensional cube is defined by the dimensional classification nodes.

After having introduced the basic concepts and fixed our terminology for the multidimensional data model, we now present a specialized technique for the design of multidimensional schemas, the ME/R modeling technique.

The reader interested in our formalization of the multidimensional data model is referred to chapter 3.5, where it is introduced as part of the FIESTA approach.

2.4. The ME/R Modeling Technique for MD Schema Design

To reflect the peculiarities of multidimensional schema design for OLAP systems, the ME/R modeling technique has been developed (together with C. Sapia) as part of the BabelFish project. We present this modeling technique as part of the common core of all BabelFish activities.

2.4.1. The ME/R Modeling Technique

The schema design methodology that has been theoretically proposed within BabelFish has also been deployed in several industrial projects, e.g. the OPAL-M project with ESG GmbH and a project with Wacker Chemie GmbH.

Core of the methodology is the ME/R modeling technique [SBHD98] for multidimensional schema design. The ME/R modeling technique is an extension of the well-known Entity/Relationship approach [Che76]. A lot of variations of the E/R model (for an overview see

e.g. [Teo94]) have been published since the first proposal of Chen. The ME/R notation uses a very basic version of the E/R model.

We formally describe our specialized E/R model using the meta modeling approach. We adhere to the four layer technique of the ISO/IRDS standard for metadata [ISO90]. The meta model of our M/ER model (according to the Dictionary Definition Layer of the IRDS) is shown in figure 2-7. The part with the white background shows the meta model of the E/R model we use as a foundation. For the purpose of describing the meta model, we make use of an extended version of the E/R model which allows the concept of generalization. This is done to increase the readability of the meta model.

Following our key considerations in [SBHD98] and [SBH99], we introduce the following specializations:

- a special entity set: '*dimension level'*,
- two special relationship sets connecting dimension levels:
  - a special n-ary relationship set: the '*fact'* relationship set and
  - a special binary relationship set: the '*classification'* relationship set.

Since the semantic concept 'dimension level' is of central importance, we introduce a special entity set for dimension levels.

To model the structure of qualifying data, we introduce a special binary relationship set: the classification relationship. It relates a dimension level A to a dimension level B representing concepts of a higher level of abstraction (e.g. city *is classified* according to country). The classification graph is defined as follows: *RG* = (E,V) with E being the finite set of all dimension levels $e_1,..,e_k$ and V = { $(e_i,e_j) \mid i \neq j \wedge 1 \leq i,j \leq k \wedge e_i$ *is classified according to* e_j }. Due to the special semantics of the classification relation, no cycles must be contained in the graph as this could lead to semantically not reasonable infinite roll-up paths (e.g. day is classified according to month and month is classified according to day). This means the following global integrity constraint must be fulfilled ($\rightarrow$* denotes the transitive closure of the *classification relation*):

$$\forall\ e_i, e_j \in E : e_i \rightarrow^* e_j \Rightarrow i \neq j$$

Thus, the classification graph *RG* is a directed acyclic graph (DAG). The name attribute of the classification relation set describes the criteria of classification. (e.g. 'lives in' for the classification relationship set connecting 'customer' and 'geographical region').

The fact relationship set is a specialization of a general n-ary relationship set. It connects n different dimension level entities. Such a relation represents a fact (e.g. vehicle repair) of dimensionality n. A description of the fact is used as the name for the set. The directly connected dimension levels are called *atomic* dimension levels.

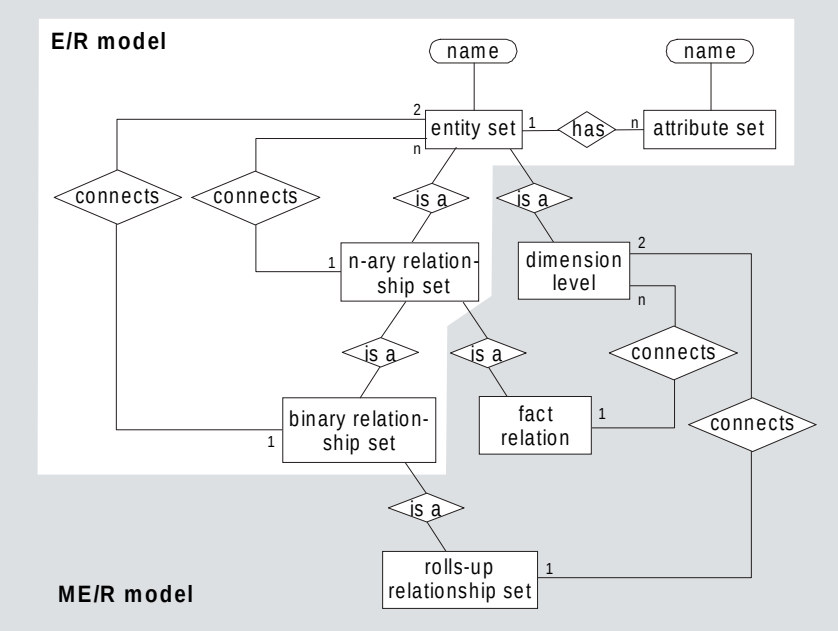


figure 2-7 : the ME/R meta model as an extension of the E/R meta model

The fact relationship set models the inherent separation of qualifying and quantifying data. The attributes of the fact relationship set model the measures of the fact (quantifying data) while dimension levels model the qualifying data.

To distinguish our specialized elements from the native E/R modeling elements and to enhance the understandability of the graphical model, we use a special graphical notation for dimension level sets, fact relationship sets, and classification relationship sets (figure 2-8).

figure 2-8 : the graphical notation of the ME/R elements

2.4.2. Example

After having formally introduced the ME/R modeling technique, we describe an example how the technique is applied within FIESTA.

As already mentioned, for modeling of the static MD cube structure we use ME/R diagrams. A typical example for the analysis of vehicle repairs is depicted in figure 2-9:

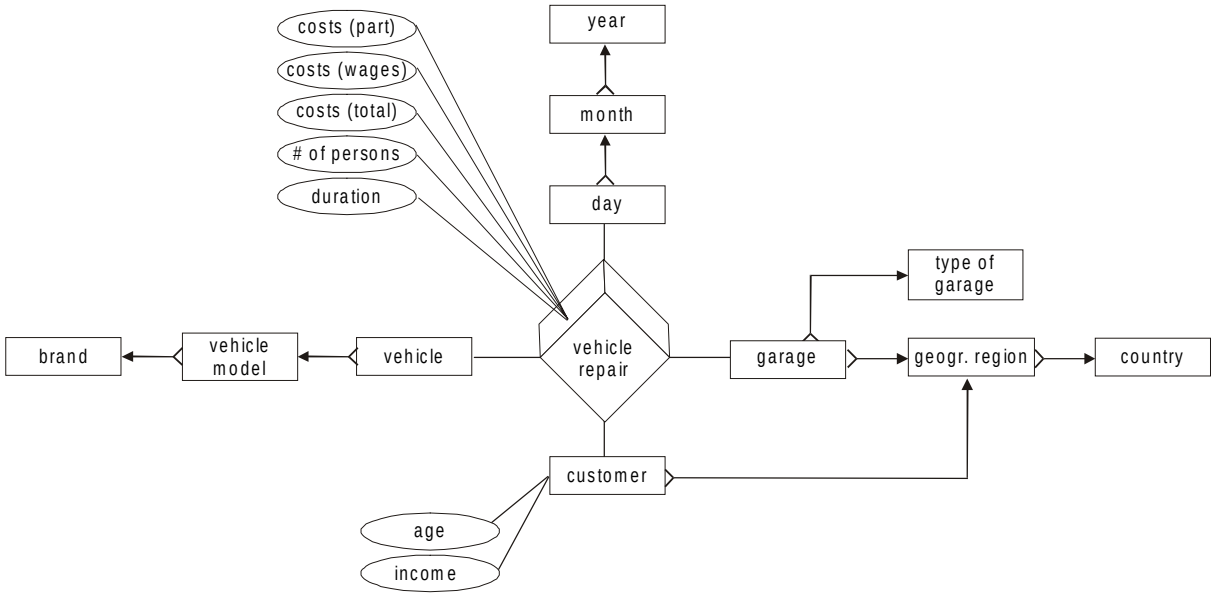


figure 2-9 : sample ME/R diagram for vehicle repairs [SBHD98]

We assume the managers of a automobile manufacturer want to analyze the repair cases for their vehicles. To this end, they model their universe of discourse as follows:

The measure data deals with vehicle repairs. We assume that a given repair is described according to the dimensions

- vehicle: the specific vehicle that had to be repaired
- day: the date of the repair
- garage: the garage that performed the repair case
- customer: the customer who owns the car.

The classification structure of the four dimensions is as follows:

- vehicles are classified according to model and brand,
- customers and garages are classified according to their geographic region and country,
- garages can also be classified according to their type (e.g. contractor or freelancer),
- and days can be classified to months and years.

For a given repair case, the following information is relevant for our scenario: the part costs, the wages, the total cost (sum of part and wages), the number of affected mechanics and the duration of the repair case.

For the dimension level customer, the describing attributes age and income have been included in the sample model.

2.4.3. The ME/R Graph Grammar: Syntax and Consistency of ME/R Graphs

Since the ME/R modeling notation is a graphical notation, one could also see a given ME/R model as a typed graph. This idea to approach a visualization of a multidimensional schema from a graph-oriented viewpoint, yields some interesting results. We will discuss some of them later (see chapter 3.6) and focus on a graph-oriented formalism here. When regarding ME/R models as typed graphs, graph grammars could be employed for the ME/R notation. The de-

tails of this approach have been worked out together with C. Sapia and K. Hahn [Hah00], [SBH00] within the BabelFish project.

We start with some basic definitions of typed graphs and graph grammars.

Definition 2-1: Typed graph

A typed graph G over a set of edge types Σ_E and a set of node types Σ_N is defined as a tuple G= (N, E, t_N, t_E, s, t) with

- N is a finite set of nodes,
- E is a finite set of edges,
- t_N: N $\rightarrow \Sigma_N$ assigns each node to its node type
- t_E: E $\rightarrow \Sigma_E$ assigns each edge to its edge type
- s,t: E $\rightarrow$ N assigns each edge to its source and target.

◊

The actual values for N, E, t_N, t_E, s, and t correspond then to the objects of a ME/R model (e.g. account, customer) whereas the definition of the edge and node types Σ_E and Σ_N (e.g. objects, relationships) is part of the modeling notation (or meta-model, in this case the ME/R notation) [SBH00].

If we regard ME/R models as typed graphs, we can use a graph grammar to describe the syntactical constraints for the modeling notation. Graph grammars are a natural means for defining the syntax of visual languages [RS97] and are typically used in graphical editors which support free graph editing and parsing of the graph structures for further processing. Since this underlying idea perfectly matches our vision of a tool-supported environment for schema design and maintenance, we adopt the concept of graph grammars for the use within FIESTA and BabelFish. Consequently, we define a graph grammar for ME/R graphs that represent an MD schema.

Definition 2-2: Graph Grammar

A graph grammar over a set of edge types Σ_E and a set of node types Σ_N is defined as a tuple (d_0, P) with

- d_0 is a nonempty initial typed graph over (Σ_E, Σ_N) called the axiom
- P is a finite set of productions. Each production p is of the form L$\rightarrow$ R, where L and R are typed graphs over (Σ_E, Σ_N) with L being the left side and R being the right side. The replacement of non-terminals in graphs is far more complicated than in linear texts. Therefore, different embedding strategies have been proposed to solve this problem. [Ehr79]. We use the concept of contexts [SBH00]. This means that both sides of the production contain a common context graph that allows for defining to which part of the existing graph the new elements should be connected.

◊

The parsing problem for context sensitive grammars is in general intractable. Therefore, we restricted our approach to layered graph grammars as presented in [RS97] to allow parsing ME/R graphs without restricting the expressiveness to context-free grammars.

The first version of the ME/R graph grammar is shown in figure 2-10 and allows for a description of syntactically correct ME/R models. Here, the embedding problem is solved by defining embedding points representing the context in the production rules (gray shaded elements in the figure). All gray shaded parts in the left side of a production rule represent parts of the graph that have a connection with the inserted partial graph after the usage of that rule. These parts are again represented as gray shaded elements in the right side of the production rule.

The axiom is represented by rule 1 and creates a fact node with two dimensions (for pragmatic reasons, we define a model to be multidimensional if it contains at least two dimensions). The definition of the single rules is quite straightforward, consequently, we omit a detailed explanation of the single rules and refer to [Hah00] for details.

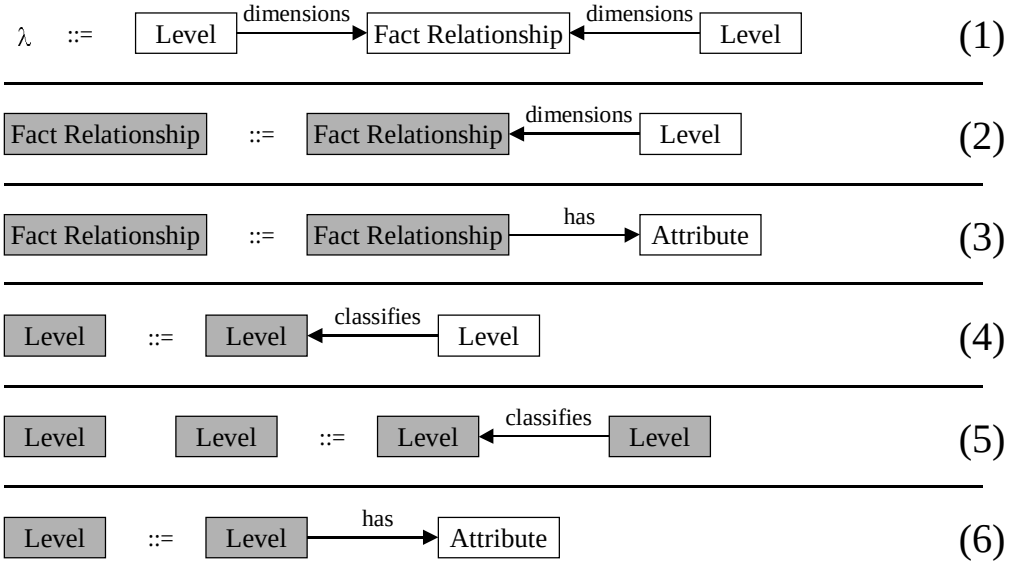


figure 2-10 : graph grammar for ME/R models

This version of the ME/R graph grammar still has some shortcomings because on one hand, it allows for semantically wrong models and on the other hand, not all practically reasonable models can be created by using the defined production rules:

- rule 5 allows to create cycles of dimension level nodes and classification relationships in a dimension (figure 2-11). The classification graph of a dimension must be a directed, acyclic graph. A check of this consistency rule is part of the semantics and therefore not possible within the formalism of a graph grammar.

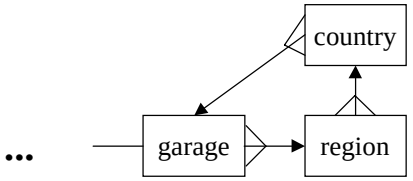


figure 2-11 : semantically incorrect cycle in a ME/R graph

- there is no possibility to create models with more than one fact node although this is quite common in real world models [SBHD98].
- no rule allows fact nodes to share a dimension. This possibility is quite useful for the conceptual model because typically the time dimension is contained in every fact relationship. [SBHD98].

In order to overcome these problems, the ME/R graph grammar has been extended by the rules in figure 2-12. Rule 7 allows to connect an existing fact node with an existing dimension level node and thus the integration of a dimension into two fact relationships. Rules 8 and 9 allow for more facts in a model with the condition that a fact must have at least two dimensions.

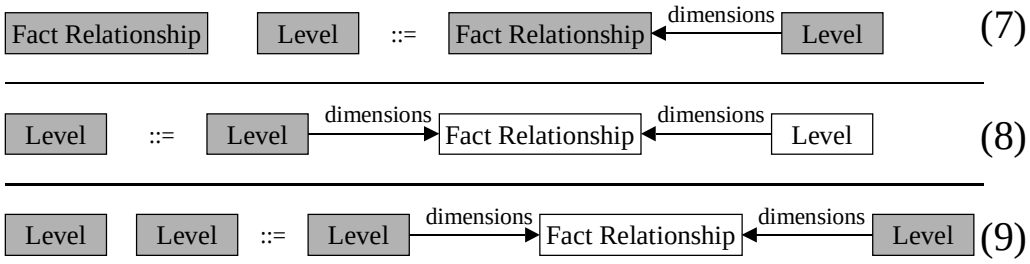


figure 2-12 : extensions of the ME/R graph grammar

The extended ME/R graph grammar has been used for the design and implementation of a syntax parser for ME/R models [Hah00]. The question of semantical consistency (e.g. cycles in the dimensions) has been excluded for the scope of this master thesis.

Checking a model for syntactical correctness corresponds to the construction of a parser for a graph grammar. This problem is far from being trivial, [RS97] presents a parsing algorithm trying to reduce the complexity using a breadth-first search and specialized filter rules for the rules not to be used at a certain point in the parsing process.

For a quite simple graph grammar like the ME/R graph grammar, a sophisticated algorithm as in [RS97] was not necessary. Using the production rules in a certain order and traversing the graph for finding the next correct rule to be used allowed for the construction of a parser without backtracking [Hah00]. The parser is a bottom-up parser, i.e., for a given model it reduces the ME/R graph by applying the production rules in reverse order. This means that a right side of a production rule identified in the graph is replaced by the expression on the left side of the rule.

If the model is correct and the rules have been applied in the correct order, the graph can be completely reduced to the empty model. Concerning the correct order of the rule appliance, a multiple traversal of a given graph is necessary. Since most models are of restricted complexity (typically less than 100 nodes), this workaround leads not to remarkable performance loss. The detailed algorithm of the parser is given in [Hah00].

2.4.4. Tool Support

Concerning tool support for the conceptual design of OLAP systems, we started a study that investigated the modeling facilities and techniques offered by commercial products [DSVH97]. This state of the art in commercial products was compared with our requirements concerning

conceptual warehouse design [DSBH98], [DSBH99]. This comparison can be seen as analogy to the state of the art in the scientific approaches for warehouse design (see chapter 1.3).

Within the BabelFish tool environment, ME/R diagrams are edited with a graphical design tool. Since the ME/R notation is still under evaluation at several industrial project partners, we decided to build a modeling tool that stores not only the conceptual models (data), but also the modeling technique itself (metadata). Both parts of information are stored in a repository [Sof98]. This decision allows for easy changes of the modeling notation because the changed modeling technique just has to be changed accordingly in the repository. Changes in the program code of the model editor are superfluous. The tool asks at startup which modeling technique to use.

The design and implementation of this generic metadata-driven graphical Modeling Tool (called GraMMi) has been developed as a master's thesis, co-supervised by the author [Haa99]. The description of the ME/R modeling technique together with the specific ME/R models is stored in Softlab Enabler, a commercial repository. Experiences with the design of the corresponding metamodels can be found in [SBH00]. A screen snapshot of the GraMMi tool can be seen in figure 2-13.

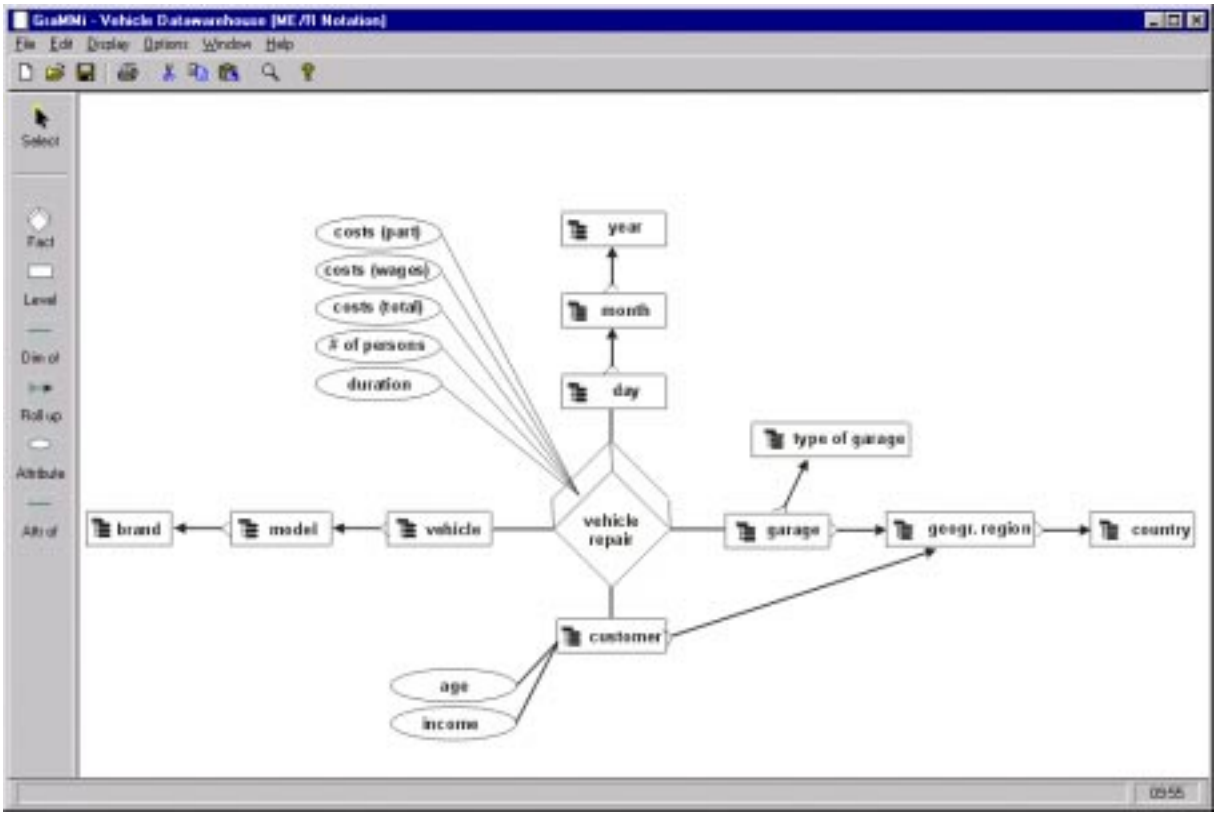


figure 2-13 : screenshot of the GraMMi tool for ME/R design

The graphical modeling tool allows the OLAP designer to build the multidimensional structure of the cube. For a running OLAP system, this multidimensional model must be mapped to a given implementation. To this end, corresponding commands must be generated for the creation of the logical database schema. As subject of a master's thesis [Hah00] (co-supervised by

the author), a specialized generator component has been designed and implemented. The generator creates schema definitions for the following commercial OLAP products:

- Cognos Powerplay ([Cog98a], [Cog98b], [Cog98c])
- Informix MetaCube ([Inf98c], [Inf98d])
- Oracle Express [Ora97]

A prerequisite for the schema generator has been a thorough investigation of the modeling capabilities and restrictions of these three commercial products. This work has been carried out as a student project [Ulb99], co-supervised by the author.

2.5. Summary

In this section we provided necessary prerequisites for the understanding of the thesis. Starting with the overall vision of the project BabelFish into which this thesis is embedded, we fixed the conceptual multidimensional data model as the central point for the design and maintenance of the OLAP system. According to our layer model, the conceptual data model is mapped to a corresponding schema in the underlying database system. The database system is also responsible for the persistent storage of the warehouse data. We call the database layer the logical layer of the architecture and regard implementations using both a relational and a multidimensional database system. The database system internal layer (called the physical layer) is not within the scope of the thesis. Here, issues like the indexing strategies (in case of a relational DBS) or tiling strategies (for a multidimensional DBS) are regarded.

A variety of formalizations for the multidimensional data model have been proposed in the literature. We contributed an overview of the elements of the multidimensional data model and fixed the terminology (which is independent of any formalization) used in this thesis.

In our tool-supported environment we use ME/R diagrams to visualize and maintain the multidimensional schema. Thus, we introduced the basics of the ME/R technique using both a metamodeling approach and a real world example. Since any graphical formalism like the ME/R technique can also be regarded as a typed graph, we presented a graph grammar for ME/R models. As an application of this graph grammar we developed a parser to detect correct ME/R models. Finally, we sketched our overall tool environment by presenting results of the BabelFish project, namely, the generic modeling tool GraMMi, its application for ME/R models, and a generation component that takes a given ME/R model as input and creates corresponding database schemas for three commercial products.

After having introduced the basics for the understanding of the FIESTA approach and important common research results of the BabelFish project, we now present the core ideas and contributions of FIESTA in the next chapter.

I must begin with a good body of facts and not from a principle (in which I always suspect some fallacy) and then as much deduction as you please.

(Charles Darwin)

3. FIESTA: An MD Schema Evolution Methodology

Evolution of MD schemas is the central subject of FIESTA. After a motivation why a methodology for multidimensional schema evolution is necessary and a valuable scientific contribution, we present an example for a schema evolution case. Next, we derive the objectives our approach has to satisfy and introduce a formalization of our approach to schema evolution. Then, we present our multidimensional data model for FIESTA and describe the dualism between ME/R models – regarded as typed graphs - and their algebraic representation – an MD schema. The heart of FIESTA, the multidimensional schema evolution operations, follow in the next section. We introduce sequences of schema evolution operations and show how to check consistency in our graphical modeling environment. Finally, we conclude with a short summary.

3.1. Motivation

As already seen in chapter 1.3, schema evolution support became a requirement with upcoming highly dynamic application domains that lead to frequent changes not only of the data itself, but also of the structure (i.e., the schema) of the data. Especially design environments like CASE and CAD systems are typically built on top of object-oriented database systems and require support for frequent schema changes from the underlying database system [Cas92], [BeLi91], [Höf96]. Another reason for the implementation on top of an object-oriented database system (OODBS) is the strong ability to reflect the peculiarities of these so-called non-standard applications [Sim95],[KhAb90]. Since the object-oriented data model contains considerable semantic complexity (e.g. the complex inheritance hierarchies) that had to be addressed by the upcoming schema evolution approaches, schema evolution became an important and still active research issue in the area of object-oriented database systems.

In addition to the state of the art shown in chapter 1.3 (which motivates why schema evolution has been thoroughly investigated for e.g. object-oriented databases), we now discuss why schema changes occur also quite often in the context of MDDBS and OLAP applications.

In recent years, enterprises launched projects aiming at building global business process models and data models covering the complete range of activities of an enterprise. However, most of these projects failed due to the often underestimated complexity and the highly dynamic structure of today's big enterprises [Dev97], [Inm96]. The experience gained from these projects was that the next generation of decision support systems started with a smaller focus, e.g. a departmental one. After having built a prototype, the scope and focus of the decision support system were extended. This iterative approach is often summarized in the motto of Bill Inmon, the so-called father of the data warehouse: "think big, start small!" [Inm96].

This approach was also applied in several data warehousing projects that FORWISS performed for industrial partners: the prototype OPAL-M for ESG GmbH and consulting services for BMW AG and Wacker Chemie GmbH. The projects started with a limited prototype which has been extended in its scope in further projects.

But even after the OLAP system is fully operational, schema changes still occur, because the typical OLAP user works directly with the multidimensional schema. This is contrary to traditional applications where the user works with an application program encapsulating the schema details of an relational or object-oriented database system. In the case of OLAP tools, the user works with the different dimensions of a fact, rolls-up and drills-down along the dimension hierarchies and selects slices/dices from his MD cube. Therefore, the user very often states new requirements to the OLAP designer concerning the MD cube structure. These new requirements have to be reflected in the MD schema leading to a constant evolution of the MD schema.

Summarizing these experiences collected both from the literature ([Inm96], [Sim95], [Dev97]) and industrial projects together with the existing research work, we draw our conclusion that both the object-oriented and multidimensional paradigm are similar with respect to the complex semantics of their data model and the often highly dynamic structure of the data (i.e., the database schema).

Regarding the existing research work (as presented in chapter 1.3), the question arises how all this existing work can be re-used and transferred to another context, e.g. to the multidimensional paradigm. Consequently, the question refines to the research issue of a common methodology for schema evolution which is independent of the underlying paradigm.

We developed such a "roadmap" to schema evolution and contribute a generic meta-model consisting of several research issues that have to be developed and problems to be solved when investigating schema evolution in general. The generic roadmap has then to be instantiated for a given paradigm (e.g. object-oriented or multidimensional). FIESTA investigates this roadmap to the case of the multidimensional data model.

Our roadmap of the schema evolution research process comprises:

1. Evolution Algebra: formal definition of the data model (schema and instances) and schema evolution operations

   The notion of a schema, in contrast to the notion of instances has to be formally defined as a prerequisite for all further steps. The same applies to the definition of schema consistency, i.e., a set of integrity rules ensuring correct schemata of the instances and a correct association of instances to their corresponding schema element. This first step has been subject to numerous publications in the area of object-oriented databases (see chapter 1.3.2).

In the case of MDDBS, there is still no common notion of MD schemas and MD instances in the literature. Existing approaches differ widely in this issue [SBH99]. We will show later how we deal with this first step in the case of multidimensional databases.

A set of schema evolution operations formally defines what types of modifications can occur on a given database schema. In addition to the syntax which e.g. defines how correct formulae (composed of MD schemas and (nested) operations) can be built, the semantics of each operation has to be defined as well. This is usually specified in terms of the data model, e.g. by set transformations. For each operation, the semantics concerning the schema (schema transformation) and existing instances (instance adaptation) must be formally defined. An important issue are the integrity constraints guaranteeing that a consistent schema is transformed to another consistent schema.

tbd <gemeinsamen Nenner consistency (relational / OO / MD) rausarbeiten>

2. Execution Model: Propagation Rules and Integrity Constraints

   Changes of a conceptual data model have to be processed in the corresponding logical data model (for definitions of the layers, see chapter 2.2). This means that the logical schema and instances have to be adapted accordingly. Formal propagation rules describe how the changes on the conceptual level are propagated to (i.e., executed in) the target (logical) environment. The question how a given conceptual (object-oriented or multidimensional) data model is implemented is a typical design decision. For both object-oriented and multidimensional applications, implementations using the same paradigm (i.e., using an OODBMS for an object-oriented data model and using an MDDBMS for a multidimensional data model) are as common as implementations on top of an RDBMS. The decision in favor of an RDBMS is often due to the proven reliability and scalability of today's commercial products. Consequently, the logical schema respectively instances may be either within the same paradigm as the conceptual schema or it may be a relational schema with relational instances.

   If an RDBMS is used for the implementation, a mapping from the paradigm of the conceptual level to the relational data model has to be defined. The formal propagation rules then describe how the requested changes on the conceptual (e.g. object-oriented or multidimensional) level are propagated to the relational schema and instances. Changes in the conceptual model would in this case be transformed to a set of SQL commands.

   But even if the logical data model is within the same paradigm as the conceptual data model, propagation rules have to be defined. The changes at the conceptual level would then generate corresponding DML/DDL commands for the deployed target system which adapt the logical schema and instances.

   Corresponding formal integrity constraints have to be defined in either case. These integrity constraints guarantee the consistency between the conceptual and the logical layer. Basically, they guarantee that all conceptual model elements are reflected in the logical database schema with the correct semantics. Examples are the foreign key relationships in a relational star schema. These constraints are the correspondence to the *is-dimension-of* relationship between a dimension (level) and a fact on the conceptual layer.

3. Software Architecture as refinement of the execution model

   Finally, there must be a precise specification how this formal framework is implemented in a given environment. Namely, the system architecture (describing different layers and possible client/server interfaces), and a component architecture (i.e., the different components with their respective functionality, interface, and input/output data) have to be specified.

This roadmap has been elaborated from existing research work and constitutes a result for present and future research in the area of schema evolution.

However, since the single steps are generic, they have to be specified and worked out with respect to the corresponding paradigm. This means that results from object-oriented schema evolution can only be applied within the object-oriented paradigm. This means further that the ideas and existing approaches from e.g. object-oriented schema evolution cannot be directly transferred to the case of multidimensional databases because of the different semantics of the two underlying paradigms (e.g. inheritance hierarchies play an important role in the object-oriented paradigm whereas dimensions exist only in the multidimensional paradigm).

Speaking in terms of schema and instances, we know the "schema" for multidimensional schema evolution (i.e., our roadmap), but we still need an "instantiation" of this meta-model specifically designed for the multidimensional paradigm. This approach would then reflect the peculiarities of the multidimensional data model and could be used to approach schema evolution for the case of multidimensional databases.

Such an approach is the subject and scientific contribution of this thesis. We call our approach FIESTA (Framework for Schema Evolution in Multidimensional Databases).

3.2. MD Schema Evolution Example

According to our basic design principles introduced in chapter 2, the work of FIESTA starts when the ME/R model (reflecting the multidimensional schema) has to be changed by the OLAP designer due to new or changed user requirements. These modifications of the static multidimensional cube structure constitute schema evolution jobs. The question arises how these evolution jobs can be processed, i.e., mapped to the underlying logical schema and instances. This task shall be performed automatically by our tool environment. Of course, the efficient execution of schema evolution jobs is an important requirement and consequently the efficient processing subject to optimization techniques. We will present our findings concerning efficient execution and optimization in chapter 4.

As an example for an MD schema evolution case, we assume the following ME/R model for the analysis of vehicle repairs as introduced in chapter 2.4.2, but in an earlier version. The difference is that the vehicle dimension consists only of the level *vehicle*. This is depicted in figure 3-1:

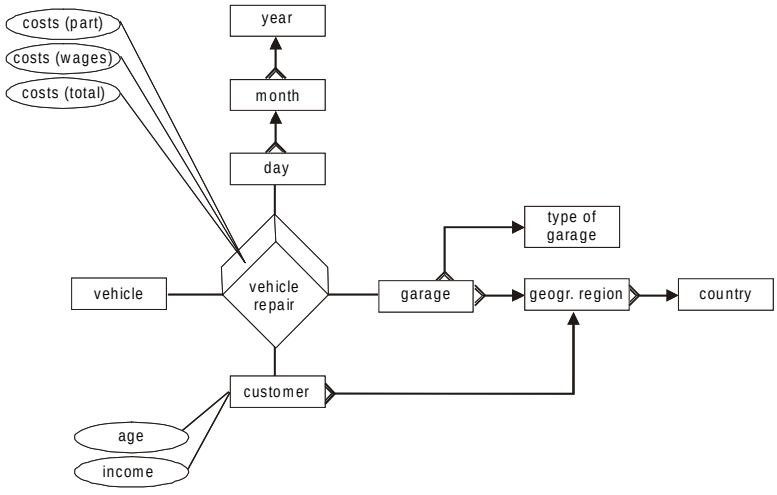


figure 3-1 : ME/R model for vehicle repairs before schema evolution

In order to show an example of an evolution job, we assume that we want to add the dimension levels *vehicle model* and *brand* to the vehicle dimension (which leads to the complete ME/R model as introduced in chapter 2.4.2). This extended ME/R model is depicted in figure 3-2. The grey shaded parts on the left side show the new dimension levels *vehicle model* and *brand* together with the corresponding rolls-up relationships.

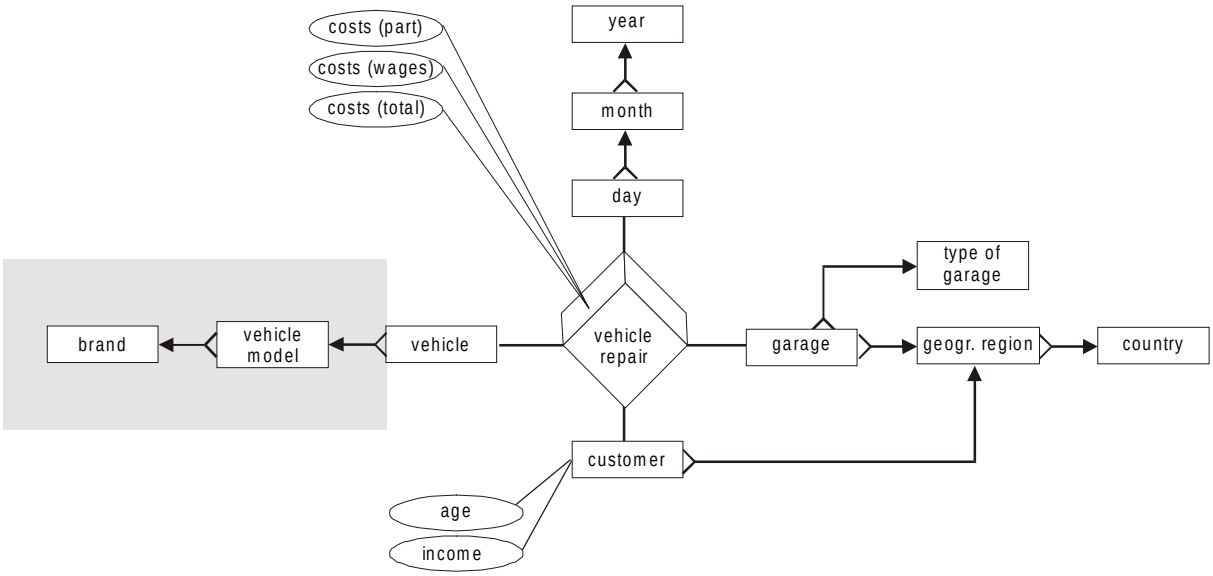


figure 3-2 : ME/R model for vehicle repairs after schema evolution

The evolution job is then specified by the OLAP designer on the ME/R graph using our graphical modeling tool GraMMi (see chapter 2.4.4). This means that it is expressed by operations like adding or deleting an edge or a node. Since ME/R models only visualize a multidimensional schema, FIESTA defines a set of 14 conceptual evolution operations that clearly define the semantics of schema evolution cases, based on our multidimensional data model. The GraMMi tool for schema design and maintenance records these modifications of the ME/R model and generates corresponding evolution jobs (defined on the algebraic representation of the ME/R model) which are then stored in the FIESTA repository for further processing.

In our example, the generated evolution job looks as follows[2]:

(1) add level (vehicle model)

(2) add level (brand)

(3) add classification relationship (vehicle, vehicle model)

(4) add classification relationship (vehicle model, brand)

The evolution jobs are then processed by a specialized evolution component that reads these evolution jobs from the repository and generates commands that adapt the logical schema and instances (including OLAP tool metadata) expressed in the target system (i.e., database system) DML/DDL. Concerning logical schema transformation and metadata update, the design and implementation have been done within a student project described in [Vet99]. Target system for this project is Informix Metacube ([Inf98c], [Inf98d]), a ROLAP product. Basically,

[2] In order to show the idea, we omit some details like the exact formal specification of the operations.

the evolution component generates SQL DDL/DML commands that adapt the logical (star) schema together with the instances in the Informix Dynamic Server Database ([Inf98a], [Inf98b]) and update the Metacube metadata accordingly (which is also stored in the Dynamic Server Database. This is again done by SQL DML commands). A corresponding SQL code fragment for the example above is shown in figure 3-3:

```
/* Schema transformation
ALTER TABLE VEHICLE ADD (MODEL CHARACTER);
ALTER TABLE VEHICLE ADD (BRAND CHARACTER);
/* Instance adaptation
UPDATE VEHICLE SET MODEL = ...;
UPDATE VEHICLE SET BRAND = ...;
/* Metadata update, insert level model
INSERT INTO DIM_EL ...;
INSERT INTO ATT ...;
INSERT INTO UI_ATT ...;
/* Metadata update, insert level brand
INSERT INTO DIM_EL ...;
INSERT INTO ATT ...;
INSERT INTO UI_ATT ...;
/* Metadata update, insert classification relationships
INSERT INTO ROLLUP ...;
```

figure 3-3 : generated SQL fragment for the evolution job

As a remark, we add that the generated DML commands updating the tool metadata are strongly dependent from the employed tool. For details about these generated SQL commands, we refer to chapter 4.4 or [Vet99].

After having explained the scope of an MD schema evolution case (both on the conceptual and the logical layer), we now present the objectives for our approach.

3.3. FIESTA Objectives

tbd <objectives concerning software architecture>

The overall objective of FIESTA is to introduce a framework that supports schema evolution for OLAP systems that are specified and managed on a conceptual level in a tool-supported environment. Since the schema evolution jobs are specified on the conceptual multidimensional layer, the task of FIESTA comprises an automatic adaptation of the schema and instances (including tool metadata) on the logical layer, tailored to a given implementation.

Following the BabelFish idea and taking this overall objective for FIESTA into account, we now derive detailed objectives for our work.

Some of these objectives have their origin in existing research work for schema evolution (i.e., they also constitute a partial result from our roadmap to schema evolution). They have been considered to be useful also for the case of multidimensional schema evolution and thus have been adapted for FIESTA.

The objectives may be grouped in three main areas: first, objectives concerning the FIESTA evolution algebra (multidimensional data model, evolution operations), second, objectives concerning the execution of evolution jobs, and third, an objective concerning the software architecture.

3.3.1. Objectives concerning the FIESTA Evolution Algebra

- support of the full design and maintenance cycle:

  Our evolution algebra supports all phases of the design and maintenance cycle. This means that our evolution operations are suited both for initial schema design (where no data is persistently stored in the warehouse system) and also for adaptations of a system populated with data (i.e., our evolution operations describe the schema transformation as well as the adaptation of existing instances).

- formal definition of semantics of evolution operations:

  The informal semantics of a given schema evolution operation may offer more alternatives concerning the execution. A typical example for different semantics of an evolution operation are cascading deletes. If we delete the lowest level of a dimension from a fact, there are two different possible semantics: one is to delete the complete dimension from the fact, the other is just to delete the lowest level of the dimension and use the next level in the dimension hierarchy as the base level for the fact. Additionally, if a dimension level or a fact is deleted, the question of additionally deleting the instances or keeping them (as a matter of optimization) has to be resolved.

  We allow for a explicit description of the alternatives by the definition of fine grained operations. Thus, our methodology removes ambiguities and defines the formal semantics of the evolution operations.

  The ambiguities may further extend to the management of the logical schema. For example, if we delete a level from a dimension hierarchy, it is not stated if this necessarily leads to deleting the respective attribute in the logical schema. Similarly, if a dimension is completely removed from a fact, the corresponding logical dimension table (if we assume a relational logical schema) may still be persistently kept for reasons of optimization.

- definition of fine grained schema evolution operations:

  FIESTA defines fine grained evolution operations that can be grouped to operation sequences. The user interactions can then be treated as a sequence of evolution operations. This approach does not only allow to model the different alternatives explicitly (see item above), but is also necessary when the sequence has to be processed, i.e. transformed to a set of DDL/DML commands for the schema and instance adaptation on the logical layer. Additionally, this allows for different variants of the graphical notation in the schema design tool. Finally, since a typical schema evolution session in the design tool leads to a sequence of operations, optimization techniques may be applied on this set of operations (e.g. re-ordering to reduce execution time).

3.3.2. Objectives concerning the FIESTA Execution Model

- automatic adaptation of logical schema, instances and tool metadata:

  For a given evolution job, the logical schema must be adapted accordingly. Similarly, existing instances should be adapted to the new schema automatically. The necessary rules for this adaptation (which have been defined by the evolution operations) are maintained and applied by the system. Two alternatives for the instance adaptation shall be possible: both physical adaptation (i.e., real modification of the persistent instances in the DBMS) and logical adaptation (i.e., construction of a current filter or query rewriting for the access of instances) shall be provided. Finally, the OLAP tool metadata should be adapted automatically.
- formulation and check of integrity constraints:

  In general, there is no formalism for e.g. expressing multidimensional constraints. There exist many notions of consistency on different layers in the scope of FIESTA. FIESTA allows not only for checking the (syntactical) correctness of multidimensional schemas and evolution jobs (i.e., sequences of schema evolution operations) but also the consistency between the conceptual and logical layer. Thus, we provide a formal foundation for defining and checking consistency in every arising context.

3.3.3. Objective concerning the FIESTA Software Architecture

Concerning the software architecture, there was only one specific objective arising from the BabelFish idea: the use of a repository system as central metadata management system.

Following the BabelFish idea of the repository-driven OLAP design, all metadata of the tool environment is stored in a repository system. Examples of metadata within the scope of FIESTA include a description of both the conceptual and the logical schema, the tool metadata, schema evolution operations, and information about the mapping between the different layers. Where possible, dedicated services of the repository system [BD94] (e.g. notification, versioning) are used to control the flow of data between the single software components.

Summarizing, our approach is used as a basis for tool-supported schema changes for OLAP systems. The FIESTA implementation provides an easy-to-use tool environment allowing to perform schema modifications without detailed knowledge about the specific implementation and frontend tools. The schema designer does not have to adapt different configurations and metadata of a frontend tool and a database schema which must be consistent for a given implementation, but the tool environment of FIESTA is responsible for performing the necessary steps in a consistent and semantically correct way providing a single point of control accessible via a graphical formalism.

3.4. Formal Approach to MD Schema Evolution

Tbd: andere formale Definitionen für MDDBMS zitieren, falls es welche gibt.

After having motivated our work and described the objectives for FIESTA, we now give a precise formulation of the research problem that is addressed by FIESTA. To understand this problem statement, the interrelationships between FIESTA and the BabelFish project (see chapter 2) have to be briefly revisited.

We assume that the OLAP designer performs changes of the conceptual multidimensional schema using a graphical design tool. The multidimensional schema is displayed in ME/R nota-

tion [SBHD98], a multidimensional modeling technique. These multidimensional schema changes, specified at the conceptual level by an ordered set of schema evolution operations, must be propagated (i.e., mapped) to a given implementation, i.e., the schema changes have to be executed in the underlying (logical) database which is e.g. an MDDBMS or an RDBMS. However, since the end-user works with this conceptual multidimensional schema in an OLAP tool, the tool metadata has to be adapted. This leads not only to modifications of the logical schema, but additionally, existing instances and OLAP tool metadata have to be adapted as well. The OLAP schema designer does not have to modify the logical schema, instances, and metadata himself, but FIESTA has the necessary knowledge to automatically adapt the logical schema, metadata, and logical instances with respect to the defined notion of consistency between the conceptual and logical layer.

Starting from our BabelFish layer model and abstracting from OLAP systems to a more generic architecture (which could also be used in the area of e.g. scientific and statistical databases) we want to introduce the term **Multidimensional Information System (MDIS)** for this kind of software systems. MDIS fulfill the following characteristics:

- A database system (as component of the overall architecture) stores the data in permanent fashion and offers access to this data.
- Specialized frontend tools (e.g. OLAP tools) present the data to the user using the multidimensional paradigm (e.g. by the cube metaphor). This view reflects the end-users understanding of the problem domain.
- An MDIS designer is responsible for modeling the end-users problem domain. This multidimensional conceptual schema can be designed and maintained by a graphical design tool.
- The system offers facilities to interactively manipulate and query the data using multidimensional operations (e.g. slicing and drilling).

tbd <Bild MDIS>

Typical application areas for MDIS are OLAP / Data Warehousing or Scientific & Statistical Databases. FIESTA focuses on OLAP applications and investigates specifically not the issue of querying the database with an OLAP tool, but addresses the issue of conceptual schema design and maintenance for MDIS.

Following the BabelFish idea, we use conceptual schemas to describe which classes of entities and propositions are of importance for a particular universe of discourse (UoD) of our application area (according to the role of a conceptual schema as defined in [ISO82], [Eic91]).

A conceptual schema must be able to yield benefits in the following three areas [ZaMe82]:

- data independence: the conceptual schema must provide a high degree of physical and logical data independence. This is extremely important in the area of MDIS for OLAP and data warehousing, because common design methodologies mix up design decisions (e.g. the question whether an RDBMS or an MDDBMS is used) with the conceptual multidimensional schema (i.e., the task of defining the multidimensional cube structure).
- design aid: the conceptual schema constitutes a typically graphical formalization of the required universe of discourse which is needed to get feedback from the end-users of an MDIS. Using the visualized conceptual schema, the OLAP designer is able to check if he meets the user requirements with his proposed design.
- liaison to the enterprise world: a conceptual model is often advocated as a successful means of communication between the IT department and the rest of an enterprise. Although this seems to be a rather historic argument, it regains importance in today’s quickly

changing business processes. Consequently, the BabelFish approach supports this idea by a graphical visualization of the conceptual schema.

According to our layer model introduced in chapter 2.2, we define a conceptual MDIS schema CS as a multidimensional schema. Similarly, we define the set of conceptual instances I_{cs} as the set of multidimensional instances according to the schema CS. Following our idea of a tool-based environment for conceptual design, we visualize the conceptual model as ME/R diagram (for a sample ME/R diagram, see chapter 2.4.2 or [SBHD98]). Formal definitions of multidimensional schema and instances will be given in chapter 3.5. Basically, they will be expressed by means of a multidimensional algebra similar to those introduced in chapter 1.3.4.

The logical schema is used for persistent storage of the conceptual schema in a database system. Consequently, we define a logical schema LS as a database schema (e.g. a relational database schema or a multidimensional database schema) which persistently stores the conceptual schema CS. The logical instances I_{LS} are the instances persistently stored in the DBS according to LS. For the focus of FIESTA, LS is a relational schema and I_{LS} are the corresponding relation instances.

Conceptual multidimensional schemas are implemented in an RDBMS as so-called star and snowflake schemas and their variants [Inm96], [McG96], [Kim96a], [Sir97]. Since the transformation of a conceptual multidimensional schema into a star schema looses parts of the multidimensional semantics (e.g. the information whether an attribute of a dimension level is used for the classification hierarchy or as describing information), OLAP tools store this semantics as part of the logical schema in the corresponding database. Consequently, we regard this metadata as part of the logical schema.

tbd <Vorteil: einfaches Handling, Nachteil: produktspezifisch, zumindest P.klassenspezifisch (d.h. MDDBS, RDBS)>

tbd <der Wolfi Lehner macht hier folgendes: star schema und snowflake [Kimball, Inmon, ich ergänze: McGuff, Holger] sind definiert. RM eignet sich für die persistente Ablage von MD Strukturen. Abb. von MD auf star schema ist in Gyssens/Laks beschrieben. Dort sind auch die Konsistenzbedingungen beschrieben (Fremdschlüssel etc.). und damit war's das.>

tbd <muß ich jetzt auch star/snowflake schemata formal definieren? eigentlich wird das formal definiert beim Algo für α in Kap.5 -> siehe Lehner -> Verweis auf Gyssens, inhaltlich siehe Beschreibung von α in Kap. 5!>

We now derive our layered formal model for schema evolution of an MDIS. To this end, we introduce the required concepts and definitions step by step. The overview is shown in figure 3-4 and formally described thereafter.

Definition 3-1: Conceptual state of an MDIS, conceptual consistency

We define the conceptual state of an MDIS as a tuple $\Sigma_c = \langle CS, I_{CS} \rangle$ with CS being a conceptual multidimensional schema and I_{CS} being the data (set of instances) according to the schema CS.

Let C_C be the set of multidimensional schema constraints. We define Σ_C to be **consistent** (or speak of **conceptual consistency**) iff each $c \in C_C$ holds.

◊

Formal definitions of conceptual schemas and instances within FIESTA will be given in chapter 3.5, where we introduce our multidimensional data model. The schema constraints C_C basically constitute a set of rules ensuring correct multidimensional schemas. In general, there are different types of constraints, e.g. model constraints (which basically come from the node/edge

type in an ME/R graph of a multidimensional model) or domain constraints (which are especially interesting in the case of the instance adaptation). We refer to chapters 3.5 to 3.8 for details on constraints.

Definition 3-2: Logical state of an MDIS, logical consistency

We define the logical state of an MDIS as a tuple Σ_L= <**LS, I_{LS}**> with LS being a logical (i.e., relational) schema and I_{LS} being the set of instances with schema LS.

Let C_L be the set of constraints ensuring the correctness of the logical schema. We define Σ_L to be **consistent** (or speak of **logical consistency**) iff each $c \in C_L$ holds.

◊

Examples for C_L are the foreign key constraints if the logical schema is implemented as a so-called star schema. Another example are the relationships between this star schema and the corresponding entries in the OLAP tool metadata tables.

Definition 3-3: Mapping function of an MDIS

Since the logical state of an MDIS is used for persistent storage of the conceptual state, a mapping function α is defined that maps the conceptual state w.r.t. a set of design decisions D to a logical state. Formally: $\alpha: \Sigma_C \text{ x } D \rightarrow \Sigma_L$

◊

Informally, the design decisions constitute e.g. the different possibilities of the mapping between the multidimensional and relational layer (e.g. star vs. snowflake schema or the different alternatives for modeling the dimension tables [Sir97], [McG97]).

According to this definition, α is a generic mapping function from the multidimensional data model to the relational data model. As we will see later (in chapter 4.1), α has to satisfy certain integrity constraints to guarantee the correctness of the mapping from a given multidimensional schema to a relational schema (including tool metadata).

Definition 3-4: Consistency of an MDIS

An MDIS with a conceptual state Σ_C, a logical state Σ_L and a mapping α from the conceptual to the logical state is defined to be consistent iff

$\sum_C$ *is consistent with* $\sum_L$ *under* $\alpha :\Leftrightarrow$

(1) Σ_C is consistent

(2) Σ_L is consistent

(3) $\alpha : \sum_C \times D \rightarrow \sum_L$ is defined in a way that each integrity constraint $c \in C_\alpha$ holds.

◊

The set of mapping constraints C_α ensures the correctness of the mapping from the conceptual to the logical layer. We will formally introduce the set C_α in chapter 4.1.

Definition 3-5: State of an MDIS

The state Σ of an MDIS is defined as $\Sigma := <\Sigma_C , \Sigma_L , \alpha>$

◊

Definition 3-6: Conceptual Schema Evolution

A conceptual schema evolution is an ordered set of operations $\gamma = (co_1, co_2, \ldots, co_n)$ with each co_i: $\sum_C^i \rightarrow \sum_C^{i+1}$, i=1, ..., n-1, where $\sum_C^i$ denotes the state before the evolution operation co_i and $\sum_C^{i+1}$ the state after the evolution operation co_i.

We also say that $\gamma(< CS, I_{CS} >) = co_1 \circ co_2 \circ \ldots co_n(< CS, I_{CS} >) = < CS', I'_{CS} >$ or $\gamma(\sum_C) = \sum'_C$ when speaking of a given schema evolution job.

◊

Examples for conceptual schema evolution operations are given in chapter 3.7. Informally, they consist of modifications of the corresponding ME/R graph (e.g. add dimension to fact, insert classification relationship, insert dimension level) which represents the multidimensional schema and its components.

Definition 3-7: Logical Schema Evolution

A logical schema evolution is an ordered set of operations $\lambda = (lo_1, lo_2, \ldots, lo_n)$ with each lo_i : $\sum_L^i \rightarrow \sum_L^{i+1}$, i=1, ..., n-1, where $\sum_L^i$ denotes the state before the evolution operation lo_i and $\sum_L^{i+1}$ the state after the evolution operation lo_i.

We also say that $\lambda(< LS, I_{LS} >) = co_1 \circ co_2 \circ \ldots co_n(< LS, I_{LS} >) = < LS', I'_{LS} >$ or $\lambda(\sum_L) = \sum'_L$ when speaking of a given schema evolution job.

◊

For the scope of FIESTA, logical schema evolution operations are e.g. sequences of SQL DML/DDL commands that adapt the logical schema and instances (including the OLAP tool metadata).

Example: tbd <Grafik bzw. bei Monika extrahieren>

Definition 3-8: Consistency of MDIS Evolution

An MDIS Evolution that transforms the MDIS from the state $\Sigma := <\Sigma_C, \Sigma_L, \alpha, IC>$ to the state $\sum' := < \sum'_C, \sum'_L, \alpha, IC >$ by using a conceptual schema evolution γ and a logical schema evolution λ is defined to be consistent iff

(1) Σ_C is consistent with Σ_L under α and IC

(2) $\sum'_C$ is consistent with $\sum'_L$ under α and IC

(3) $\gamma(\sum_C) = \sum'_C$ and

(4) $\lambda(\sum_L) = \sum'_L$

◊

tbd Carsten: was is consistent under α?

The given definitions are necessary prerequisites for explaining figure 3-4 and deriving a formal problem statement for FIESTA (figure 3-5). The overall evolution scenario for an MDIS is depicted in the following figure 3-4.

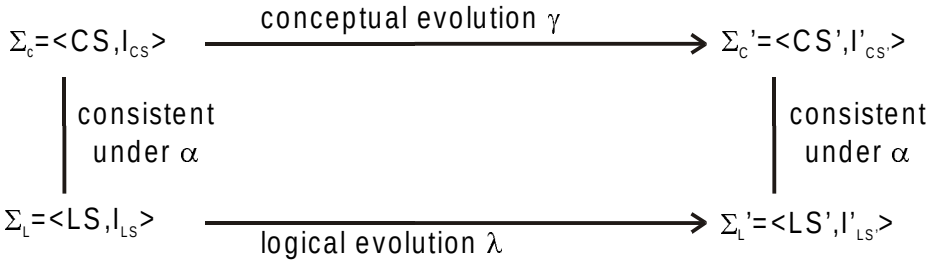


figure 3-4: FIESTA schema evolution scenario for MDIS

We assume the OLAP designer works with the conceptual multidimensional schema of the MDIS. For this schema, instances (cells of the cube) can be defined accordingly, but the OLAP designer works only with the conceptual multidimensional schema, because he does not need the conceptual instances for his schema design task. The conceptual state before the evolution is depicted as Σ_C = <CS, I_{CS}>, the state after the evolution as Σ_C'= <CS',$I'_{CS'}$> (as defined above).

We further assume that according to our definition of an MDIS, we have a persistent storage in an DBS on the logical OLAP layer. The logical state before the evolution is depicted as Σ_L=<LS, I_{LS}>, the state after the evolution as Σ_L' = <LS',$I'_{LS'}$>.

The function α maps between the conceptual and logical layer as defined above and ensures consistency between the two layers. The evolution is depicted both on the conceptual level (by γ) and the logical level (by λ).

For defining the research problem of FIESTA, we assume the following:

Given: CS and γ are given[3] (resulting from the schema design work in our graphical design tool which shows CS' after the necessary transformations are done), LS and I_{LS} are persistently stored in the database system. Σ_C is consistent with Σ_L.

Required: the logical evolution λ that adapts our persistent storage of the logical schema and instances and is consistent with γ w.r.t. α.

This is depicted in figure 3-5. The grey shaded parts are given, the logical evolution λ is required and leads to the target Σ_L'. The constraint that the target has to fulfill is the consistency with Σ_C' resulting in a consistent MDIS evolution.

[3] CS and CS' allow for a computation of γ (except for the order), CS' and γ let us derive CS. Our graphical schema design environment stores CS and records γ.

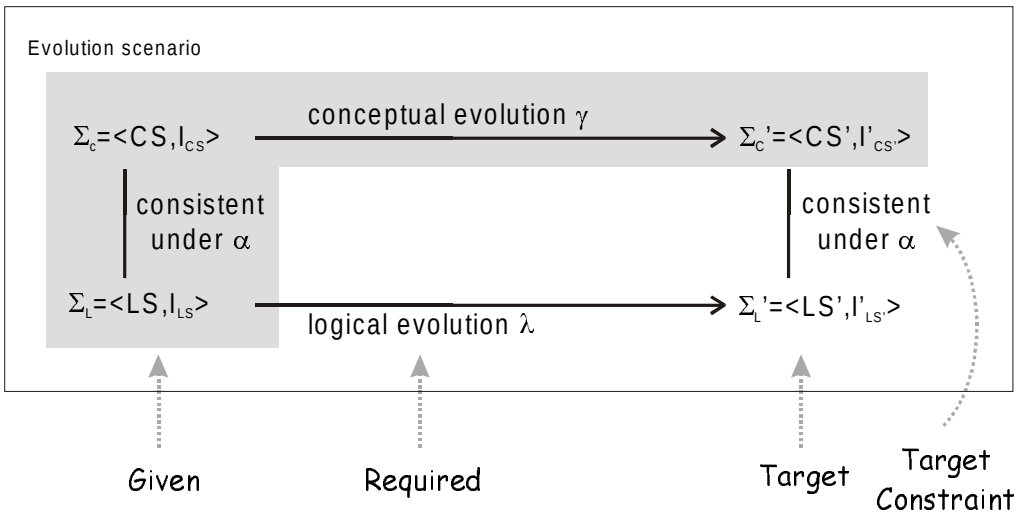


figure 3-5: FIESTA schema evolution problem description

Formally, we may define

Definition 3-9: FIESTA Schema Evolution Problem

We assume a consistent MDIS with state $\Sigma := <\Sigma_C , \Sigma_L , \alpha>$ and a set of consistent conceptual schema evolution operations $\gamma = (co_1, co_2, ... , co_n)$ that transform Σ_C to Σ_C'. We require the set of logical schema evolution operations $\lambda = (lo_1, lo_2, ... , lo_m)$ that transform Σ_L to Σ_L' such that Σ_C' is consistent with Σ_L' under α.

◊

Note that for the computation of λ we need the following data as input: Σ_C, Σ_L, γ, α and our notion of consistency defined by C_c, C_L, C_α. Basically, we transform the set of operations γ using the other data as additional input and compute λ. The detailed algorithm for this computation is presented in chapter 4.4.

Up to now, we have not yet detailed the definition of a conceptual schema CS and the according set of instances I_{CS}. In chapter 1.3.4 we presented several formalizations of the multidimensional data model that have been discussed in the literature. This state of the art already introduced formal definitions of a multidimensional schema and instances. In the next section, we present our multidimensional data model that has been developed together with C. Sapia in the BabelFish project.

3.5. Multidimensional Data Model

As a prerequisite for defining multidimensional schema evolution operations, formal definitions of multidimensional schemas and instances have to be provided. To this end, we introduce our multidimensional data model (or, more precisely: our formalization of the multidimensional data model). Our survey [BSHD98], [SBH99] of the existing approaches (see chapter 1.3.4) revealed several advantages and disadvantages of variant formalizations. On this base, we developed our own data model for FIESTA with a special focus on a comprehensive and easily understandable definition of our schema evolution operations (which will be introduced in chapter 3.7).

The multidimensional data model has been published in [BSH99] together with a first version of the schema evolution operations.

3.5.1. Requirements to a formal multidimensional data model

Before introducing our formal multidimensional data model, we start with a set of requirements that an ideal multidimensional data model should fulfill. These requirements are derived from general design principles that have proven successful with the relational model and from characteristics of OLAP applications we have developed for our industrial partners.

The requirements constitute to some degree an extension to chapter 2.3 (where we introduced the basic terminology for the multidimensional paradigm) and have already been introduced in [BSHD98], [SBH99]. Since their first introduction, they have proven their value by numerous citations.

We introduce three groups of requirements: general requirements for a formal multidimensional data model (the first three enumerated items), requirements concerning complex structured dimensions, and complex structured cube cells:

- **Implementation independent formalism:** The formal model must be purely conceptual, thus not containing any details of the implementation. This is especially important in the area of OLAP applications as some existing systems (ROLAP systems) implement multidimensionality by mapping the conceptually multidimensional model to a relational model.
- **Separation of structure and contents:** The formalism should allow the separated definition of the data structure (i.e. the multidimensional cube and its dimensions) and the contents (i.e. the cell values).
- **Declarative query language:** Analogous to SQL, the multidimensional query language should be declarative to allow query optimization and data independence. A logical calculus or an algebra allowing optimizations are considered declarative for this purpose. Since the main focus of our research work is schema evolution and not query processing, we do not concentrate further on that issue.
- **Complex structured dimensions:** dimensions provide the context information about the data that is to be analyzed. Technically speaking, the dimensions of a cube span the multidimensional space. In classical arrays the dimensions of the multidimensional data space are only structured by a linear order defined on the indexes (typically integer values). For OLAP applications this is not sufficient because from the view of the OLAP end-user, the elements (respectively instances) of an OLAP dimension (dimension members) are not linearly ordered (e.g. garages). Instead, hierarchies containing dimension levels are used for the structuring of a dimension (see also chapter 2.3).

  Another way of structuring dimensions from a user's point of view is the use of dimension attributes. These attributes describe dimension members but do not define hierarchies (e.g. it might be meaningful to store the name and address of a customer). Different levels of the hierarchies can possess different dimension attributes.

  Hierarchies and attributes structuring dimensions are part of the schema of the database and it should not be necessary to include the structural or functional definition in any query. As OLAP analysis is characterized by a high degree of interactivity it should nevertheless be possible for the user to define an ad-hoc hierarchy when querying the database (e.g. for a single query, a user wants to classify vehicles by price which is not modeled in the schema).

- **Complex structured cell values (measures):** The contents of a cell of the multidimensional cube can also be structured in a complex way. Each cell can contain several measures that form a record structure. OLAP applications often contain a considerable amount of derived measures. These are measures that are not atomic in the sense that they can be computed from other measures (atomic or derived) in the cube. Depending on the calculation formula, derived measures can define hierarchies on atomic measures.

  The treatment of complex measures in the context of aggregation is also interesting. The computation of aggregation functions might not be semantically meaningful for all the measures. E.g. the summarization of "the number of persons" that participated in a repair along the time dimension does not produce semantically meaningful results. On the other hand, an aggregation using the *sum* operator along the garages dimension is sensible, as is the computation of the average number of persons involved in a repair (that corresponds to an aggregation along the time dimension using the *avg*-operator). Such constraints (often referred to as additivity of a measure along a dimension) should be expressible in the conceptual model.

  The concept of derived measures is analogous to the view concept of relational systems. Thus, the definition of derived measures (calculation formula) should be a part of the schema of the database. Derived and atomic measures should be treated equally by the query language. Nevertheless, the query language should also support ad-hoc calculations defined within the query.

These requirements are fulfilled to different degrees in the existing formal multidimensional data models (see our survey and chapter 1.3.4). Consequently, for the objective to propose schema evolution operations, we have developed a formal multidimensional data model which is especially suited for a comprehensive and easily understandable definition of schema evolution operations. This FIESTA multidimensional data model is now introduced.

3.5.2. Multidimensional Schema

As mentioned above, several interpretations of the multidimensional paradigm can be found both in the existing literature (e.g. [CT98], [DT97], [DKPW99], [Leh98], [Vas98]) and in product implementations. A comparison of the formal approaches shows that most of them do not formally distinguish between schema and instances [SBH99], as their main goal is a formal treatment of queries using algebras and calculi. For our research work, we need a formalism that can serve as a basis for defining the schema evolution operations (see chapter 3.7). Therefore, this section contains a formal definition of a multidimensional schema and its instances (which was inspired by the formal multidimensional models mentioned above, esp. [CT98], [DT97], [Vas98]).

Since we require a clear separation between schema and instances, we provide separate constructs. From the approaches examined in chapter 1.3.4, only [LW96], [GL97], [CT98] and [Leh98] explicitly make this distinction, too.

The schema (or MD model) of an MDIS contains the structure of the facts (with their attributes) and their dimension levels (with their attributes) including different classification paths (that reflect the hierarchical structure of the dimensions).

Definition 3-10: Alphabet, Character Sequences:

We assume a finite alphabet Z and denote the set of all finite sequences over Z as Z*.

◊

Before formally introducing our definition of an MD schema, we explain and introduce the components informally. We will define three distinct sets to model facts, dimension levels and attributes. Facts constitute the subject of the analysis, i.e. ‘sales’ or ‘repairs’ are typical examples for facts. Dimension levels reflect the elements of dimension hierarchies (see chapter 2.3 for the terminology). Attributes may be either measures of a fact or describing attributes of a dimension level. We will introduce a dedicated function that assigns an attribute either to a fact or to a dimension level. To reflect the dimension hierarchies, we will introduce a relation on the level names that relates two levels by a corresponding classification hierarchy. Finally, we define a function that assigns the base dimension levels (i.e. the finest granule of the dimensions) to the corresponding fact, e.g. for our vehicle repair example, the base levels of the four dimensions are *day*, *vehicle*, *customer* and *garage*. We remark that this function represents the structural relationship between the fact and its “lowest” or “finest” dimension levels. This structural relationship has to be seen independently from the possible aggregations which are typically of interest when processing queries. Of course, the measures can also be queried at a higher dimension level than the base level (e.g. vehicle repairs by month, customer, and geographic region of the garage). The calculation of the necessary aggregation is not within the scope of FIESTA.

Definition 3-11: MD Model, MD Schema:

An MD model (or MD schema)[4] $\mathcal{M}$ is a 6–tuple <F, L, A, *gran*, *class*, *attr*> where

(1) $\mathbf{F} \subset Z^*$ is a finite set of *m* fact names $\{f_1,\ldots,f_m\}$ where $f_i \in Z^*$ for $1 \le i \le m$

(2) $\mathbf{L} \subset Z^*$ is a finite set of *k* dimension level names $\{l_1,\ldots,l_k\}$ where $l_i \in Z^*$ for $1 \le i \le k$.

(3) $\mathbf{A} \subset Z^*$ is a finite set of *p* attribute names $\{a_1,\ldots,a_p\}$ where $a_i \in Z^*$ for $1 \le i \le p$. Each attribute name a_i has a domain *dom*(a_i) attached.

(4) The names of facts, levels and attributes are all disjoint, i.e. $L \cap F \cap A = \varnothing$

(5) *gran*: $F \rightarrow 2^L$ is a function that associates a set of dimension level names with a fact. These dimension levels *gran(f)* are called the base levels of fact *f*.

(6) *class* $\subseteq L \times L$ is a relation defined on the level names. Without imposing any restrictions, we require *class* to be minimal w.r.t. transitivity. The transitive, reflexive closure *class** of *class* must fulfill the following property: $(l_1,l_2) \in class^* \Rightarrow (l_2,l_1) \notin class^*$. This means that *class** defines a partial order on L. $(l_1,l_2) \in class^*$ reads "l_1 can be classified according to l_2."

(7) *attr:* $A \rightarrow F \cup L \cup \{\perp\}$ is a function mapping an attribute either to a fact (in this case the attribute is called a measure), to a dimension level (in this case it is called dimension level attribute) or to the special $\perp$ symbol which means that this attribute is not connected at all.

◊

One bigger difference of this definition to almost all existing approaches is the fact that we have no dedicated construct for the notion of a dimension. The reason is quite simple, but possibly not obvious at this early stage of our work: we do not need such an explicit formalization of a dimension. We have the dimension levels and their classification relationships, both constructs together deliver the necessary information for a dimension.

We also explicitly allow for record structured measures, since the function *attr* may assign an arbitrary number of attributes to a fact. From the other approaches listed in chapter 1.3.4, only [DT97] and [GL97] allow more than a single measure. Similarly we also allow for describing attributes of a dimension level which is not possible in the approaches of [AGS97], [CT98], [Vas98].

Concerning the base levels for a fact, we decided to define a function instead of a simple set because it comes closer to the underlying idea of an n-dimensional space. Additionally, the function brings some basic constraints by definition which would have to be expressed by set constraints otherwise. The dedicated set for attributes has its origin in the ME/R modeling notation. Here, we wanted our algebraic formalization as close as possible to the graphical representation. This approach delivers some benefits which will be pointed out later in this thesis. Basically, it helps in defining a complete dualism between an ME/R model and the algebraic representation of an MD schema. This is further elaborated in chapter 3.6.

Example: we use again the following example taken from [SBHD98]. A car manufacturer wants to analyze vehicle repairs to improve the technical quality of its products, to evaluate the warranty policy and to assess the quality of different garages. After the first iteration of the development cycle, the model shown in figure 3-6 was implemented.

[4] We use the term MD model when we have a graphical multidimensional schema in mind, consequently, we speak of an MD schema when thinking of an algebraic representation of the schema.

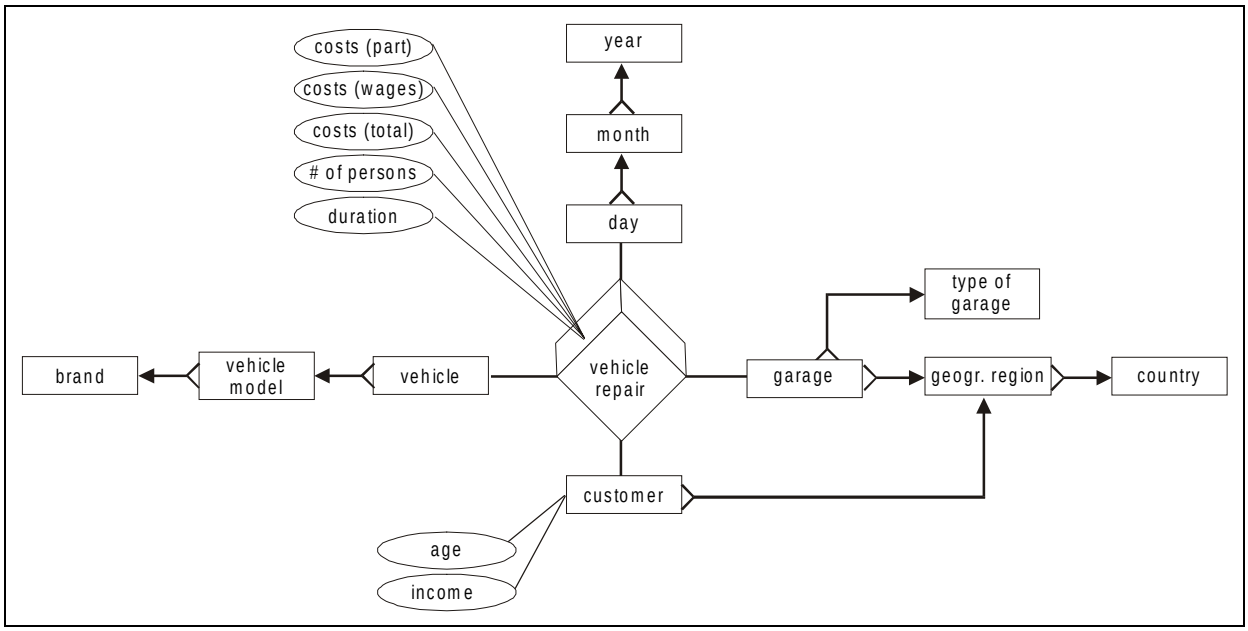


figure 3-6: graphical representation of the MD schema (using the ME/R notation)

Formally, the example MD Model $\mathcal{M}_{ex}$=<F_{ex}, L_{ex}, A_{ex}, $gran_{ex}$, $class_{ex}$, $attr_{ex}$> has the following components:
F_{ex} = {vehicle repair}
L_{ex} = {customer, vehicle, vehicle model, brand, day, month, year, garage, type of garage, geogr. region, country }
A_{ex} = {costs (part), costs (wages), costs (total), # of persons, duration, age, income}
$gran_{ex}$(vehicle repair) = {customer, vehicle, day, garage}
$class_{ex}$ = {(day, month), (month, year), (garage, type of garage), (garage, geogr. region), (geogr. region, country), (customer, geogr. region), (vehicle, vehicle model), (vehicle model, brand)}
Finally, attr is defined as follows[5]:
$attr_{ex}$("costs (parts)") = vehicle repair, $attr_{ex}$("costs (wages)")= vehicle repair,
$attr_{ex}$("costs (total)")= vehicle repair, $attr_{ex}$("# of persons")= vehicle repair,
$attr_{ex}$(duration)= vehicle repair, $attr_{ex}$(age) = customer, $attr_{ex}$(income) = customer

◊

3.5.3. Cube Instances

The MD model formalizes the schema of a multidimensional database. We use this formalism later to define a set of schema evolution operations. As we also want to analyze the effects of schema evolution operations on the instances of the schema, this section presents a formal model for instances. We start with some base definitions as prerequisites.

[5] We have used the quotation marks (") to avoid confusion with the arguments of attr, where necessary.

Definition 3-12: Domain of a Dimension Level:

The domain of a dimension level $l \in L$ is a finite set $dom(l) = \{m_1, \ldots, m_q\}$ of dimension member names.

◊

To represent the structure for the fact instances, we introduce a domain and a co-domain. The domain of a fact is the cross-product of all base dimension levels (representing the coordinates of the cube cell), whereas the co-domain is the cross-product of all measures for this fact (representing the record structure of this cube cell).

Definition 3-13: Domain and Co-domain of a fact:

For a fact *f*, the domain *dom(f)* and co-domain *codom(f)* are defined as follows:

$$dom(f) := \bigtimes_{l \in gran(f)} dom(l)$$

$$codom(f) := \bigtimes_{\{a | attr(a) = f\}} dom(a)$$

We remark that the implicit order of the cartesian products in this definition are not relevant and will consequently be ignored.

◊

In the definition of MD instances, we will introduce the following components: first, a set of functions (one for each classification) that represent the classification relationship for the elements (instances) of the two dimension levels. This formalizes the structural assignment between the instances of the two dimension levels. Next, for every fact a function that maps the coordinates of the cube to the measures, i.e. a mapping from the instances of the base levels to the measures. Finally, for every describing dimension attribute, we need a function that assigns the value of this attribute to every dimension member (instance).

Definition 3-14: Instance of MD model:

The instance of an MD model $\mathcal{M}$ = <F, L, A, *gran*, *class*, *attr*> is a triple

$\Im_{\mathcal{M}}$ = <R-UP, C, AV> where

(1) R-UP = $\{ r-up_{lev1}^{lev2} \}$ is a finite set of functions with $r-up_{lev1}^{lev2} : dom(lev1) \rightarrow dom(lev2)$ for all $(lev1, lev2) \in class$

(2) $C = \{c_{f_1}, \ldots, c_{f_m}\}$; $f_i \in F \; \forall 1 \le i \le m$ is a finite set of functions c_f: dom(f) → codom(f); $f \in F$. C maps coordinates of the cube to measures, thus defining the contents of the data cube.

(3) AV = { av_1, …, av_r } is a finite set of functions which contains a function av_a for each attribute a that is a dimension level attribute, i.e., *attr*(a) ∈ L. The function av_a: $dom(attr(a)) \rightarrow dom(a)$ assigns an attribute value (for attribute a) to each member of the corresponding level.

◊

Our definition of MD cube instances is quite different from other formalizations. The main reason for this is the proximity to our MD schema definition. Of course, we also aimed at a comprehensive and easily understandable definition of the instance adaptation for the schema evolution operations. Thus, we have to refer the reader to chapter 3.7 for a discussion of the benefits of our instance formalization.

Example (Instances):
A possible instance of the MD model $\mathcal{m}_{ex}$ is <R-UP$_{ex}$, $C_{ex,}$ AV$_{ex}$ > as defined above.
Let us assume the following domains for the levels:

dom(customer) = { “Mr. Burns”, “Mr. Simpson”};
dom(garage) = { “Springfield”, “Los Angeles”, “New York” };
dom(geogr. region) = { “USA West”, “USA East” };
dom(day) = { “01/01/97”, “01/02/97” ... };

According to our definitions, the domain and co-domain of the fact *vehicle repair* are[5]:
dom(vehicle repair) = *dom*(customer) × *dom*(garage) × *dom*(day) × *dom* (vehicle)
codom(vehicle repair)=*dom*(“costs (parts)”)×*dom*(“costs (wages)”)×*dom*(“costs (total)”)
× *dom* (“# of persons”) × *dom* (duration)

R-UP$_{ex}$ = { $r-up_{garage}^{geogr.region}$, $r-up_{garage}^{type\ of\ garage}$, $r-up_{geogr.region}^{country}$, $r-up_{customer}^{geogr.region}$, $r-up_{vehicle}^{vehicle\ model}$, $r-up_{vehicle\ model}^{brand}$, $r-up_{day}^{month}$, $r-up_{month}^{year}$ } where $r-up_{level1}^{level2}$ is a function mapping the members of level1 to members of the higher level2. Some examples are

$$r-up_{garage}^{geogr.region}(\text{“Springfield”}) = \text{“USA West”}$$

$$r-up_{garage}^{geogr.region}(\text{“New York”}) = \text{“USA East”}$$

Cube schema: C_{ex} = {$c_{vehicle\ repair}$} with
$c_{vehicle\ repair}$: *dom*(vehicle repair) → *codom*(vehicle repair).

As an example, we define
$c_{vehicle\ repair}$ (“Mr. Burns”, “Springfield”, “06/27/1998”, “car4711”) = ($500, $200, $700, 2, 8)

AV$_{ex}$ = {$av_{age,}$ av_{income} } with av_{age}: dom(customer) → dom(age) and
av_{income}:dom(customer) → dom(income)

As some examples, we may define
av_{age} (“Mr. Burns”) = 70
av_{age} (“Mr. Simpson”) = 41
av_{income} (“Mr. Burns”) = 1 M$
av_{income} (“Mr. Simpson”) = 25 k$

◊

In contrast to other formalizations of the multidimensional data model (as those introduced in chapter 1.3.4), we do not propose special multidimensional operations (like slice, dice, pivot) working on the instances. These operations are of interest when regarding queries on the multidimensional data model. Since multidimensional queries are not within the scope of FIESTA, we do not introduce instance operations here.

Nevertheless, FIESTA also defines multidimensional operations, but the operations work with the multidimensional schema, not the instances. These schema evolution operations will be introduced in chapter 3.7.

For a complete formal multidimensional data model, there is still a missing issue: the formal definition of consistency. This is subject to the following chapter.

3.5.4. MD Schema Integrity Constraints

In general, multidimensional schemas described by our algebra can be inconsistent. For example, it is possible to define "empty" facts, i.e. facts without dimension attached or isolated dimension hierarchies that have no relationship to a fact. Also, since we want to introduce schema evolution operations that may lead to temporarily inconsistent schemas, we have to provide a formal notion of consistent multidimensional schemas.

Consequently, we define the following consistency constraints for multidimensional schemas.

A consistent MD schema $\mathcal{M}$ = <F, L, A, gran, class, attr> must fulfill the following constraints:

(1) every fact must be connected to at least one dimension level:

$\forall$ f $\in$ F: gran(f) must be well-defined and |gran(f)| $\geq$ 1

(2) every dimension level must be part of a classification hierarchy or connected to a fact (or in other words: isolated dimension levels must not exist):

$\forall$ l $\in$ L: ($\exists$ f $\in$ F with l $\in$ gran(f)) $\vee$ ($\exists$ x $\in$ L with either (l,x) $\in$ class or (x,l) $\in$ class)

(3) every attribute must be connected to either a fact or a dimension level

$\forall$ a $\in$ A: attr(a) must be well-defined, attr(a)$\neq\perp$ and |attr(a)| =1

(4) finally, we do not allow isolated dimension hierarchies that are not connected to a fact:

$\forall$ l $\in$ L: ($\exists$ f $\in$ F with l $\in$ gran(f)) $\vee$ ($\exists$ m $\in$ L, $\exists$ g $\in$ F with m $\in$ gran(g) $\wedge$ (m,l) $\in$ class*)

Basically, constraint (4) is an extension of constraint (2). Since class* is the reflexive and transitive closure of class, we could also combine constraints (2) and (4) to the condition (2') which would then replace both conditions:

(2') $\forall$ l $\in$ L: $\exists$ m $\in$ L, $\exists$ g $\in$ F with m $\in$ gran(g) $\wedge$ (m,l) $\in$ class*

For the rest of this chapter, we will either use the minimal version of the constraints or the extended version depending on the intuitiveness we need.

After having introduced our formalization of the multidimensional data model together with a definition of consistent multidimensional schemas, we now show the relationship between a given ME/R model and its counterpart, the algebraic representation of a multidimensional schema.

3.6. The Dualism on the Conceptual OLAP Layer: ME/R Graphs and MD Schemas

We show that the ME/R graph of a given multidimensional data model can be equivalently expressed by a multidimensional schema using the algebra defined in chapter 3.5.

Since an algebraic description of a given MD schema is typically difficult to handle, we now want to show that both the algebraic description and the ME/R representation of a given MD schema can be used equivalently, at least under some conditions that can be easily fulfilled.

Informally, we already used this dualism between ME/R graphs (as introduced in chapter 2.4.3) and our formal MD algebra in the example of chapter 3.5. There we showed our sample ME/R diagram and presented an algebraic description of the visualized MD schema.

Now, we want to formalize this relationship and introduce an isomorphism between ME/R graphs and algebraic MD schemas. To this end, we will prove that both representations of an MD can be transformed into the other, without any loss of information.

In our graphical modeling environment (see chapter 2.4), we use ME/R models to visualize an MD schema. In chapter 2.4.3 we have introduced a graph grammar for checking the correctness of these ME/R models by parsing.

Here, we follow a different, graph-oriented approach for the correctness. We consider ME/R models as typed graphs and present certain criteria for their correctness. Then we prove that an ME/R graph satisfying these criteria can be transformed into a consistent MD schema (using our algebraic MD data model). We go even further and prove that this transformation is an isomorphism between an ME/R graph and the algebraic schema representation. When transforming a correct ME/R graph to an MD schema, its consistency is guaranteed by the isomorphism. In other words, the correctness of the ME/R graph assures certain conditions of the MD schema. Later, in chapter 4, we will exploit these fulfilled conditions on the MD schema for the processing of our schema evolution operations.

But now back to the isomorphism between ME/R graphs and MD schemas. We start with the definition of ME/R graphs.

3.6.1. ME/R graphs

As the first step, we show that every ME/R graph can be equivalently expressed by an algebraic description of the MD schema. Since the ME/R modeling notation is a graphical notation, one could also see a given ME/R model as a typed graph. Typed graphs have been defined in chapter 2.4.3 (Definition 2-1). We just repeat the contents of the definition here.

A typed graph G over a set of edge types Σ_E and a set of node types Σ_N is defined as a tuple $G= (N, E, t_N, t_E, s, t)$ with

- N is a finite set of nodes,
- E is a finite set of edges,
- t_N: $N \rightarrow \Sigma_N$ assigns each node to its node type
- t_E: $E \rightarrow \Sigma_E$ assigns each edge to its edge type
- s,t: $E \rightarrow N$ assigns each edge to its source and target.

The actual values for N, E, t_N, t_E, s, and t correspond then to the objects of a ME/R model (e.g. account, customer) whereas the definition of the edge and node types Σ_E and Σ_N (e.g. objects, relationships) is part of the modeling notation (or the meta-model, in this case the ME/R notation) [SBH00].

For ME/R graphs, we define:

Definition 3-15: ME/R graph

An ME/R graph G = (N, E, t_N, t_E, s, t) is a typed graph over a set of edge types Σ_E and a set of node types Σ_N defined as

Σ_N := {dim_level, fact, attribute} for the three different types of nodes and

Σ_E := {is_classified_by, is_dimension_of, is_measure_of, is_attribute_of} for the different types of edges.

◊

In the example of figure 3-6, a possible ME/R graph definition would be:

N ={customer, region, ... , day, month, year, ...} for the set of nodes and

E = {coststotal_repair, vehicle_brand, day_month, month_year, customer_repair, ...} for the set of edges.

The set of edges could also be just a set of numbers, one for every edge. Here, we used a textual description of the two connected nodes with an underscore ("_"). There is no condition how the elements of E must look like.

The function t_N must assign the value 'fact' to every fact node, the value 'dim_level' to every dimension level node, and the value 'attribute' to every attribute node in the ME/R model. Thus, in the example, we have:

t_N(customer)= dim_level, t_N(vehicle_repair)= fact, t_N(cost_total)= attribute, ...

Similarly, t_E assigns the corresponding edge types to the edges. Each edge between two dimension levels is assigned the value 'is_classified_by', each edge between a fact and its base level is assigned the value 'is_dimension_of', each edge between an attribute and a fact is assigned the value 'is_measure_of', and each edge between an attribute and an dimension level is assigned the value 'is_attribute_of'. As examples, we enumerate

t_E(coststotal_repair)= is_measure_of, t_E(day_month)=is_classified_by,

t_E(customer_repair)= is_dimension_of, ...

Finally, the functions s and t assign the start and end node to every edge. Consequently, in our example, we have:

s(customer_repair)=vehicle_repair, t(customer_repair)=customer

3.6.2. Correctness of ME/R graphs

So far, the definition of the ME/R graph is quite open. Especially, there are no constraints how the types of edges and nodes may be assigned. We use the basic understanding of the edge and node types from the definition of the ME/R notation, but still have not defined any correctness criteria for ME/R models or graphs yet.

Thus, we propose the following criteria for the correctness of an ME/R graph:

1) no isolated nodes exist, every edge connects exactly two nodes.
2) the types of edges and nodes are assigned correctly (w.r.t. the definition of ME/R models)
3) every node is reachable from a fact by a sequence of arbitrary edges and nodes of arbitrary type.
4) there exist no cycles of *is_classified_by* edges between dimension levels.

We explain these informal conditions before we clearly define them formally for ME/R graphs.

Condition (1) simply requests that no nodes without edges or edges connected only to a single node exist. We admit that this is a restriction, but we think that a valid ME/R model simply contains no isolated elements. Discussing whether a partial graph describing only a dimension hierarchy (which is not attached to a fact) is already a correct ME/R model, is a rather academic discussion in our opinion.

Condition (2) is very important and delivers a simply expressible, but very powerful condition. Its satisfaction guarantees a lot a variants of incorrect edge types, e.g. not two attribute nodes can be connected, no classification edge connects a fact with a dimension level and so on.

Condition (3) guarantees that each partial ME/R graph is completely connected and contains at least a fact. This condition prohibits ME/R models consisting only of a dimension hierarchy or of some dimension levels with attributes. On the other hand, it explicitly allows for different partial ME/R graphs with a fact relationship as the “heart” of every partial graph. In the formalization for ME/R graphs, we will extend this condition to two sub-conditions: first (condition 3-1), the condition that a minimal ME/R graph consists of a fact node and a dimension level node, connected by an is_dimension_of edge. Second (condition 3-2), every node is reachable by a sequence of nodes and edges from a fact node as starting point.

Condition (4) avoids cycles in the classification hierarchy of a dimension. As we will see later, this is a necessary prerequisite for our normalization of ME/R graphs (see chapter 3.6.3). Cycles in the classification hierarchy would also neglect the partial order within a classification hierarchy and are thus prohibited.

For our formalization, we need a predicate *path (a,b)* that evaluates to TRUE iff there is a path from node *a* to node *b* in the corresponding ME/R graph. Formally, we define:

path (a,b) with $a,b \in N$ is TRUE iff

$(\exists\ e \in E$ with $s(e)=a$ and $t(e)=b) \vee$

$(\exists\ (n_1, n_2, \ldots, n_m)$ with $n_i \in N$, $m>0$ and $\exists\ (e_1, e_2, \ldots, e_{m+1})$ with $e_i \in E$:

$s(e_1)=a$, $t(e_1)=n_1$,

$s(e_2)=n_1$, $t(e_2)=n_2$,

.....

$s(e_m)=n_{m-1}$, $t(e_m)=n_m$,

$s(e_{m+1})=n_m$, $t(e_{m+1})=b)$

otherwise, path (a,b) is FALSE.

The conditions for ME/R graphs are formalized as follows:

(1) every node is either source or target of at least one edge:

$\forall\ n \in N: \exists\ e \in E$ with either $s(e)=n$ or $t(e)=n$

(2) every edge connects exactly two nodes and the correctness of the edge and node types is guaranteed:

$\forall\ e \in E: \exists\ n,m \in N$ with $s(e)=n$ and $t(e)=m$,

if $t_E(e)=$ is_classified_by then $t_N(n)$ = dim_level and $t_N(m)$ = dim_level

if $t_E(e)=$ is_dimension_of then $t_N(n)$ = fact and $t_N(m)$ = dim_level

if $t_E(e)=$ is_measure_of then $t_N(n)$ = attribute and $t_N(m)$ = fact

if $t_E(e)=$ is_attribute_of then $t_N(n)$ = attribute and $t_N(m)$ = dim_level

(3-1) $|\{n \in N$ with $t_N(n)=fact\}| \geq 1$ and $\forall\ f \in N$ with $t_N(f)=fact$: there is at least one $e \in E$ with $t_E(e)=$ is_dimension_of, $s(e)=f$, $t(e)=n$ for an $n \in N$ with $t_N(n)=$ dim_level.

(3-2) $\forall\ n \in N$ with $t_N(n)=$ dim_level or $t_N(n)=$ attribute:

$\exists\ f \in N$ with $t_N(f)=$ fact and path (f,n).

(4) $\forall\ n \in N$ with $t_N(n)=$ dim_level: path (n,n) = FALSE.

After having provided a means for correctness of ME/R graphs, we see that we need another prerequisite for our dualism between ME/R graphs and MD schemas: a normal form for ME/R graphs which prevents us from two different graph representations of the same MD schema.

3.6.3. Normalization of ME/R graphs

After having addressed the issue of correctness criteria for ME/R graphs, we have to deal with a different problem: the issue of uniqueness of an ME/R graph which leads us to a normal form for ME/R graphs. The basic problem are redundant edges in the classification hierarchy of a dimension. These edges do not constitute cycles in the classification hierarchy because of the direction of the classification edges. They may exist because the classification relationships are transitive. Since the schema designer may not be aware of the transitivity, he may add these redundant edges to make the ME/R model reflect his universe of discourse. The following example shows an redundant edge between day and year in the time dimension:

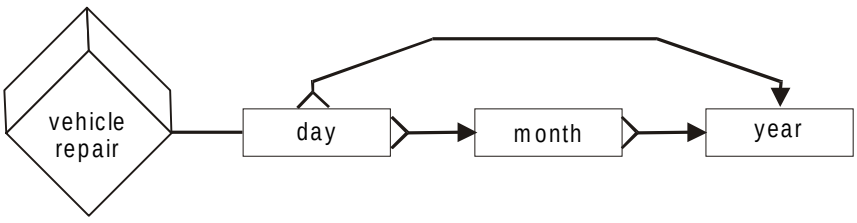


figure 3-7: redundant edge in the time dimension

As shown in figure 3-7, a redundant edge constitutes a loop (not a cycle) without intermediate nodes (i.e. of length 1) in the ME/R graph.

Redundant edges lead to the observation that there are several syntactically different ME/R graphs representing the same semantics (i.e. the dimension hierarchy). This would constitute a problem for our idea of an isomorphism between ME/R graphs and MD schemas, because the *class* relationship of an MD schema must always be minimal by definition. Therefore, we could have different ME/R graphs for the same MD schema.

Consequently, we define normalized ME/R graphs to resolve this problem. When working with ME/R graphs in our graphical modeling environment, we simply include a normalization step which removes redundant edges before further processing of an ME/R graph.

Definition 3-16: Normal Form of ME/R graph, Normalization of ME/R graphs

A correct ME/R graph $G = (N, E, t_N, t_E, s, t)$ is defined to be in *normal form* iff

$\forall\ e \in E$ with $t_E(e)$= is_classified_by and $s(e)=a$ and $t(e)=b$ and $a \neq b$:

$\neg\exists\ [(n_1, n_2, \ldots, n_m)$ with $n_i \in N$, $m>0$,

$(e_1, e_2, \ldots, e_{m+1})$ with $e_i \in E$, $e \neq e_i$, $e_i \neq e_j$ for $i \neq j$ and $t_E(e_i)$= is_classified_by] with:

$s(e_1)=a$, $t(e_1)=n_1$,

$s(e_2)=n_1$, $t(e_2)=n_2$,

.....

$s(e_m)=n_{m-1}$, $t(e_m)=n_m$,

$s(e_{m+1})=n_m$, $t(e_{m+1})=b$

The *normalization* of a correct ME/R graph $G = (N, E, t_N, t_E, s, t)$ removes all redundant edges. Formally, we define the new set of edges E' as

$E' := E \setminus \{ e \in E$ with $t_E(e)$= is_classified_by $\wedge$

$\exists\ [(n_1, n_2, \ldots, n_m)$ with $n_i \in N$, $m>0$,

$(e_1, e_2, \ldots, e_{m+1})$ with $e_i \in E$ and $t_E(e_i)$= is_classified_by] with:

$s(e)=a$ and $t(e)=b$,

$s(e_1)=a$, $t(e_1)=n_1$,

$s(e_2)=n_1$, $t(e_2)=n_2$,

.....

$s(e_m)=n_{m-1}$, $t(e_m)=n_m$,

$s(e_{m+1})=n_m$, $t(e_{m+1})=b$ }

◊

Theorem 3-1: Existence and Uniqueness of normalized ME/R graph

To every correct ME/R graph G = (N, E, t_N, t_E, s, t) there exists a normalized ME/R graph G'=(N, E', t_N, t_E', s', t') which is uniquely determined.

◊

Proof 3-1: Existence and Uniqueness of normalized ME/R graph

The existence proof is a direct consequence of the definition of normalized ME/R graphs: either the ME/R graph is already in normal form or construct G' by removing the redundant edges.

When it comes to the uniqueness of the normalized ME/R graph, we see that the correctness criterion (4) which prevents cycles in the classification hierarchy, is useful here. Let us assume the following situation for a classification hierarchy of a dimension:

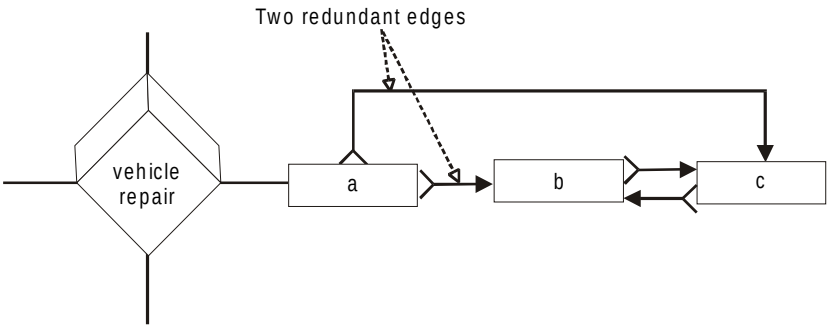


figure 3-8: two redundant edges in a dimension

In this example, there exist two different paths from a to c and from a to b: c can be reached from a either directly or via b. Similarly, b can be reached from a either directly or via c. Thus, we have two redundant edges. If we would remove the edge from a to b, we would get a different normalized graph as if we would remove the edge from a to c. This result would be in contrast to the uniqueness of the normalized ME/R graph.

This example revealed the only case where the uniqueness of the resulting normalized ME/R graph could be endangered. The cycle between b and c (represented by the two classification edges) in this incorrect ME/R graph lead to the situation where we had two conflicting redundant edges. Consequently, since cycles are prohibited in a correct ME/R graph, we conclude that the resulting normalized ME/R graph is always uniquely determined.

q.e.d. ◊

For reasons of completeness, we have to prove that a correct ME/R graph is still correct after normalization:

Theorem 3-2: Normalization and Correctness of ME/R graphs

An ME/R graph G = (N, E, t_N, t_E, s, t) which is correct according to conditions (1), (2), (3-1), (3-2), (4) is still correct after normalization.

◊

Proof 3-2: Normalization and Correctness of ME/R graphs

We omit the full formal proof. Basically, there are two cases: if G is already in normal form, it is not transformed at all and therefore still correct. If G is being normalized, only *redundant* edges are removed so that all conditions still hold (no isolated elements, correct types of edges, existing paths are not deleted and condition (3-1) is not endangered by normalization)

q.e.d. ◊

As a consequence of this theorem for the rest of this thesis, we only speak of an ME/R graph and mean the corresponding *normalized* ME/R graph.

3.6.4. Mapping ME/R graphs to MD schemas

Now we show that a typed graph for a given ME/R model satisfying conditions (1) to (4) is equivalent to an MD schema:

Theorem 3-3: Mapping ME/R graphs to MD schemas

Every typed, normalized ME/R graph G = (N, E, t_N, t_E, s, t) satisfying conditions (1) to (4) can be mapped to an equivalent consistent MD schema $\mathcal{M}$=<F, L, A, gran, class, attr>.

◊

Proof 3-3: Mapping ME/R graphs to MD schemas (by construction)

We build the MD schema as follows:

$F := \{n \in N \text{ with } t_N(n) = fact\}$, $L := \{n \in N \text{ with } t_N(n) = dim\_level\}$

$A := \{n \in N \text{ with } t_N(n) = attribute\}$

For the construction of *gran*, we define:

$\forall f \in F$: define $gran(f) := \{ n \in N \text{ with } t_N(n) = dim\_level \wedge (\exists e \in E: s(e) = f \wedge t(e) = n \wedge t_E(e) = is\_dimension\_of) \}$

$class := \{ (l_1,l_2) \text{ with } l_1,l_2 \in L \wedge t_N(l_1) = dim\_level \wedge t_N(l_2) = dim\_level \wedge (\exists e \in E: s(e) = l_1 \wedge t(e) = l_2 \wedge t_E(e) = is\_classified\_by) \}$

The minimality of class and its transitive closure (especially the partial order defined by class) is ensured by condition (4) (no cycles in the dimension hierarchy) and the normalization.

For the construction of *attr*, we define:

$$\forall a \in A : define\, attr(a) := \begin{cases} f & if\ \exists e \in E : t_E(e) = is\_measure\_of \wedge s(e) = f \wedge t(e) = a \\ l & if\ \exists e \in E : t_E(e) = is\_attribute\_of \wedge s(e) = l \wedge t(e) = a \\ \perp & if\ \neg\exists e \in E : s(e) = a \vee t(e) = a \end{cases}$$

After the construction of $\mathcal{M}$, we have to show that it fulfills the four constraints for a consistent MD schema (see chapter 3.5.4).

Condition (1) for correct MD schemas says that every fact must be connected to at least one dimension level:

$\forall$ f $\in$ F: gran(f) must be well-defined and |gran(f)| $\geq$ 1

gran(f) is well-defined by construction and conditions (1) to (3-1) for correct ME/R graphs guarantee that there is at least one dimension level connected to the fact, i.e. there is at least one base dimension level for the fact.

Condition (2) for correct MD schemas requests that every dimension level must be part of a classification hierarchy or connected to a fact (or in other words: isolated dimension levels must not exist):

$\forall$ l $\in$ L: ($\exists$ f $\in$ F with l $\in$ gran(f)) $\vee$ ($\exists$ x $\in$ L with either (l,x) $\in$ class or (x,l) $\in$ class)

This is ensured by conditions (1) and (3-2) for correct ME/R graphs.

Condition (3) for correct MD schemas requests that every attribute must be connected to either a fact or a dimension level:

$\forall$ a $\in$ A: attr(a) must be well-defined, attr(a)$\neq\perp$ and |attr(a)| =1

This is guaranteed by the conditions (1), (3-1) and (3-2) for correct ME/R graphs.

Finally, condition (4) for correct MD schemas prohibits isolated dimension hierarchies that are not connected to a fact:

$\forall$ l $\in$ L: ($\exists$ f $\in$ F with l $\in$ gran(f)) $\vee$ ($\exists$ x $\in$ L, $\exists$ g $\in$ F with m $\in$ gran(g) $\wedge$ (m,l) $\in$ class*)

This condition is satisfied by condition (3-2) in conjunction with conditions (1) and (2) for correct ME/R graphs.

q.e.d. $\Diamond$

3.6.5. Mapping MD schemas to ME/R graphs

Since the mapping between ME/R graphs and an MD schema should be bijective, we also prove the reverse direction, i.e. for every consistent MD schema there exists a correct ME/R graph:

Theorem 3-4: Mapping MD schemas to ME/R graphs

Every consistent MD schema $\mathcal{M}$=<F, L, A, gran, class, attr> can be mapped to an equivalent typed ME/R graph G = (N, E, t_N, t_E, s, t) satisfying the ME/R graph conditions (1) to (4).

$\Diamond$

Proof 3-4: Mapping MD schemas to ME/R graphs (by construction)

Assume we have a consistent MD schema $\mathcal{M}$=<F, L, A, gran, class, attr>, as defined in chapter 3.5.4. We now construct a typed ME/R graph (with Σ_E and Σ_N as defined above for ME/R graphs) G = (N, E, t_N, t_E, s, t) as follows:

The set of nodes is simply the union of all MD schema ‘nodes’:

$N := F \cup L \cup A$

The edge elements are tuples in this construction algorithm. This constitutes no restriction and is merely used for notational convenience:

E:= { (l_1,l_2) with $l_1,l_2 \in L$ and $(l_1,l_2) \in$ class } /* all classification edges */

$\cup$ { (f,l) with f ∈ F, l ∈ L and gran(f)=l } /* all is_dimension_of edges */

$\cup$ { (f,a) with f ∈ F, a ∈ A and attr(a)=f } /* all measure edges */

$\cup$ { (l,a) with l ∈ L, a ∈ A and attr(a)=l } /* all attribute edges */

The type functions are defined as:

$$\forall n \in N \ define\ t_N(n) := \begin{cases} \text{dim\_}level & if\ n \in L \\ fact & if\ n \in F \\ attribute & if\ n \in A \end{cases}$$

and

$$\forall (e_1,e_2) \in E \ define\ t_E(e_1,e_2) := \begin{cases} is\_classified\_by & if\ (e_1,e_2) \in class \\ is\_\text{dim}ension\_of & if\ gran(e_1) = e_2 \\ is\_measure\_of & if\ attr(e_2) = e_1 \wedge t_N(e_1) = fact \\ is\_attribute\_of & if\ attr(e_2) = e_1 \wedge t_N(e_1) = \text{dim\_}level \end{cases}$$

Finally, we have to define the source and the target function. These definitions are quite straightforward because we already have this information in our edge tuples. Thus, we define:

$\forall (e_1,e_2) \in E \ define\ s(e_1,e_2) := e_1$

$\forall (e_1,e_2) \in E \ define\ t(e_1,e_2) := e_2$

After the construction of the ME/R graph, we have to prove that it fulfills the conditions for correct ME/R graphs:

Condition (1) for correct ME/R graphs demands that every node is either source or target of at least one edge:

This condition is satisfied by the constraints (1) to (3) for consistent MD schemas.

Condition (2) for correct ME/R graphs requests that every edge connects exactly two nodes and the correctness of the edge and node types is guaranteed.

This condition is fulfilled by the definition of the functions gran, class and attr of the MD schema together with constraints (1) to (3).

Condition (3-1) for correct ME/R graphs demands: ∀ f∈F: there is at least one e ∈ E with t_E(e)= is_dimension_of, s(e)=f, t(e)=n for an n ∈ N with t_N(n)= dim_level.

This condition corresponds to constraint (1) for consistent MD schemas.

Condition (3-2) for correct ME/R graphs demands:

∀ n∈N with t_N(n)= dim_level or t_N(n)= attribute:

∃ f ∈ N with t_N(f)= fact and path (f,n).

To guarantee this powerful condition, we basically need all constraints for consistent MD schemas because constraint (4) only ensures that no isolated dimension hierarchies exist. For the complete path from a fact or a dimension level, we need also the dimensionality of the fact (constraint (1)), together with the connection criteria for all attribute and dimension level nodes.

Finally, the minimality condition of *class* and its transitive closure guarantees that no redundant edges exist and that there are no cycles in the dimension hierarchy (condition 4).

Consequently, the resulting ME/R graph is in normal form.

q.e.d. ◊

3.6.6. Isomorphism between ME/R graphs and MD schemas

The two mappings between ME/R graphs and MD schemas together with the normalization step allow for a full isomorphism between correct ME/R graphs and consistent MD schemas.

We omit the complete formal proof here and only present the argument chain which provides our reader with more intuitive understanding of the proof. The single arguments strongly rely on the formal proofs of the mappings and the normalization, thus the proof of the isomorphism basically would constitute a rather lengthy and straightforward extension of the three proofs above.

In order to proof the isomorphism between correct ME/R graphs and consistent MD schemas, we have to show that the mapping of a correct ME/R graph to a consistent schema is

(1) injective: if two ME/R graphs have been mapped to the same MD schema, the graphs must be identical. The only case where two (or more) different ME/R graphs would be mapped to the same MD schema, is prohibited by the normalization. Thus, using our normalization step, we could show that the mapping is injective

(2) surjective: to show this, we have to prove that there exists a mapping from consistent MD schemas to correct ME/R graphs (which we have defined in fact) and that the concatenation of the two mappings delivers the identity. In other words, a consistent MD schema, mapped to a correct ME/R graph, mapped again to a consistent MD schema would deliver the identical MD schema. This can be proven by applying our two mappings (together with the normalization) accordingly, except for different namings of the MD schema components.

3.6.7. Discussion and conclusions drawn from the dualism

Although the isomorphism between an ME/R graph and its corresponding MD schema is quite powerful, an interesting issue is not resolved (and can not be resolved) on this level. This may cause problems when further processing an ME/R graph or MD schema. The issue is concerned with **merging dimensions**, i.e. a classification relationship between dimension levels belonging to different dimensions. For the task of conceptual design, this must be allowed (e.g. it makes sense to classify both customers and garages according to their geographical region). Nevertheless, no commercial tool is powerful enough to capture this semantics accordingly (see chapter 4.1).

As a result of this further core piece of FIESTA, we may use both formalizations of an MD schema equivalently, either the ME/R model (ME/R graph) or the algebraic description of the MD schema. This isomorphism is extremely useful, because the graphical formalism fits to our approach of a schema design and maintenance tool, whereas the algebraic representation can be used for internal representation and processing. Additionally, the semantics of a schema evolution operation must be described in terms of the algebraic data model (especially for the instance adaptation).

For our graphical design tool this means that we may use the ME/R graph as interface to the user's (i.e. the schema designer) interaction. The isomorphism guarantees a consistent MD schema if we have a correct ME/R graph. The consistency of the MD schema will be further exploited when we process the schema evolution operations and generate corresponding commands for the adaptation of the logical schema and instances (see chapter 4).

3.7. Evolution of MD Schemas

tbd <Klassifikation der Evo-Ops gemäß der oben eingeführten Klassifikation (was gehört zu welchem Bestandteil des MD Data Model)>

After having provided two necessary prerequisites, namely

- a formal definition of multidimensional schemas and instances (chapter 3.5), and
- the dualism of a conceptual multidimensional schema described by both its ME/R and algebraic representation (chapter 3.6),

we are now able to present a set of formal evolution operations for multidimensional schemas.

The schema evolution operations of FIESTA have been first introduced in [Bla99], [DSBH99] and as a complete formal version in [BSH99].

Regarding our objectives introduced in chapter 3.3 and the overall idea of doing schema design in a graphical modeling tool using the ME/R representation of an MD schema, we have decided to define very fine-grained schema evolution operations. The operation definition is also closely related to our formalization of the multidimensional data model and to the elements of an ME/R model (or the ME/R graph, respectively).

This fine-grained approach yields the following benefits:

- easy use in a graphical modeling tool: since the definition of the evolution operations is close to the graphical representation of MD schema elements using the ME/R notation, the operations are close to the ME/R graph operations. Basically, the schema evolution operations work with the typed edges and nodes of an ME/R graph. Consequently, we can derive the complete set of evolution operations just by the basic graph operations:
  - inserting or deleting an ME/R node: since we have three special ME/R nodes (dimension level, fact, attribute) which we can insert or delete from an ME/R graph, we already have six operations.
  - ME/R edges: there are four different edge types in the ME/R approach: the attribute relationship between a dimension level and an attribute, the attribute relationship between a fact and an attribute, the classification relationship between two dimension levels, and the is_dimension_of relationship between a fact and its base dimension levels. The graph operations connect and disconnect for these four edge types deliver eight further operations.

  Summarizing, when regarding merely the typed ME/R graph with its specialized nodes and edges and the basic operations of inserting and deleting nodes and edges, we already have

the set of 14 operations. These graph operations constitute exactly the set of schema evolution operations working on an MD schema. Consequently, the proposed set of evolution operations will be complete in the sense that every correct ME/R graph (or consistent MD schema) can be "constructed" using the schema evolution operations (will be proven in chapter 3.8).

- The fine-grained approach allows to reflect different variants of the semantics of a schema evolution operation. For example, it is possible to express a sequence of operations that deletes a dimension from a fact together with the whole dimension hierarchy (i.e. all dimension levels together with the corresponding classification relationships). In other use cases, this semantics may not be feasible. The fine-grained approach allows to define arbitrary sequences of schema evolution operations explicitly reflecting these different semantics.
- Prerequisite for processing schema evolution operations in the target database system (on the logical layer): only the fine-grained approach allows to derive corresponding DDL/DML commands according to the features of the target system. When processing conceptual schema evolution operations on the logical layer, the fine-grained schema evolution operations have to be grouped together. The grouping rules are different for different target systems. Each grouping is then transformed to a corresponding set of DML/DDL commands in the target system. Further, the fine-grained approach allows for optimization of sequences of schema evolution operations. Details on these issues will be given in chapter 4.

As already mentioned, the fine-grained approach typically leads to a sequence of schema evolution operations. Since the single operations do not always guarantee the consistency of an MD schema after their execution, we only check for consistency after an evolution session. Chapter 3.8 further elaborates the issue of operation sequences and consistency.

Coming back to the atomic (or base) operations, we now present key ideas of our formalization.

First of all, we introduce all our operations using the same layout and description. We start with the name and a textual description of the operation. Then, we define pre- and post-conditions for the execution. Finally, we formally define the syntax with input and output parameters and the semantics of the operation execution. To this end, figure 3-9 shows a sample template for the definition of the syntax and semantics.

name of the operation	
syntax with input and output parameters	operation (par_1, par_2, ... , par_n) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_{\mathcal{M}}$, par_1, par_2, ... **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{\mathcal{M}'}$
semantics	**Schema:** formal description of $\mathcal{M}'$ **Instances:** formal description of $\mathfrak{I}'_{\mathcal{M}'}$

figure 3-9: template for the description of a schema evolution operation

The table heading contains the name of the operation, whereas the two table lines present the syntax and semantics, respectively. The syntax is defined like a function call, i.e. the operation name together with all input parameters in brackets. The first two input parameters are always the current schema and the current set of instances, followed by additional parameters like a new dimension level to be inserted. The output of an operation is always the new schema (after the execution of the operation) and the new set of instances.

Regarding the semantics of the execution of an operation, we present a formal description (in terms of our multidimensional data model) of both the schema transformation and the instance adaptation.

Appendix A provides a tabular enumeration of the complete definitions of all evolution operations.

Formally, a schema evolution operation *op* transforms an MD schema $\mathcal{M}$ =<*F, L, A, gran, class, attr*> to an MD schema $\mathcal{M}'$ =<*F', L', A', gran', class', attr'*>. Some operations also require an adaptation of the instances $\mathfrak{I}_{\mathcal{M}}$ to $\mathfrak{I}'_{\mathcal{M}'}$. We always denote elements *before* the operation execution with the regular letter (e.g. L), whereas a letter with an apostrophe (e.g. L') denotes the corresponding element *after* the operation execution.

Annotation: When a relation or function like *class* or *attr* changes only its definition or result set (and not the mapping of the elements itself), we say that e.g. class' = class and omit the strict formal definition including the definition set and result set.

For a function f:*dom*→*codom* let $f|_{dom'}$ denote the restriction of f to *dom'* $\subseteq$ *dom*

We start with the operations describing modifications of a dimension level:

3.7.1. Modification of a dimension level

1. **insert level**: this operation extends an existing MD model by a new dimension level. The operation extends the set of levels without changing the classification relationships, thus creating an isolated level element. Classifications relationships for this new level have to be defined separately.

   Precondition: the new level may not be contained in the MD model yet.

   Postconditions: after the application of the operation, the new level is part of the new MD model. The new level has no instances because we regard the definition of instances as part of the definition of corresponding classification relationships for this level.

   Syntax and semantics:

insert_level	
syntax with input and output parameters	insert_level ($\mathcal{M}$, $\mathfrak{I}_{\mathcal{M}}$, l_{new}) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_{\mathcal{M}}$, new level name l_{new} **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{\mathcal{M}'}$
semantics	**Schema:** $\mathcal{M}'$=<*F, L', A, gran', class', attr'*> *L' := L* $\cup$ { l_{new} } *gran'*:= *gran* (see annotation above) *class'* := *class*

	attr':= *attr* **Instances:** No effects on instances because the operation inserts a new and empty dimension level without instances. Thus: $\mathfrak{I}'_{m'}$ = <R-UP, C, AV>

figure 3-10: syntax and semantics of the *insert_level* operation

2. **delete level**: deletes an existing, but isolated (i.e. not connected to any other element of the MD model) dimension level l_{del} from an MD model. Instances are deleted automatically together with the dimension level..

   Preconditions: the dimension level exists ($l_{del} \in L$), the level must not be connected to a fact ($l_{del} \notin gran(f)\ \forall f \in F$) or via classification relationships ($(l_{del}, l) \notin class \wedge (l, l_{del}) \notin class\ \forall l \in L'$). Further, the level must not have any attributes attached ($attr(a) \neq l_{del}\ \forall a \in A$).

   Postconditions: after the application of the operation, the level not contained in the MD model. Existing instances are deleted.

   Syntax and semantics:

delete_level	
syntax with input and output parameters	delete_level ($\mathcal{M}, \mathfrak{I}_m, l_{del}$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, level name l_{del} to be deleted **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}'$=<*F, L', A, gran', class', attr'*>. L' := L \ { l_{del} } *gran'*:= *gran* *class'* := *class* *attr'* := *attr* **Instances:** no effect because dimension members are deleted automatically. Thus: $\mathfrak{I}'_{m'}$ = <R-UP, C, AV>

figure 3-11: syntax and semantics of the *delete_level* operation

3.7.2. Modification of an attribute

3. **insert attribute**: creates a new attribute without attaching it to a dimension level or fact. The assignment of the attribute to a dimension level or fact constitutes a separate operation. Especially, it is not defined if the new attribute is a measure or a dimension level attribute.

Precondition: the new attribute name may not be part of the existing MD model ($a_{new} \notin A$)

Postcondition: after the application of the operation, the new attribute is part of the MD model. The new attribute has no values associated.

Syntax and semantics:

insert_attribute	
syntax with input and output parameters	insert_attribute ($\mathcal{M}, \mathfrak{I}_m, a_{new}$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, attribute a_{new} with dom(a_{new}) to be inserted **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}' = \langle F, L, A', gran, class, attr' \rangle$ $A' := A \cup \{ a_{new} \}$ $attr': A' \rightarrow F \cup L \cup \{\perp\}$; $attr'(a) := attr(a)\ \forall\ a \in A' \setminus \{ a_{new} \}$, $attr'(a_{new}) := \perp$ **Instances:** no effect, thus: $\mathfrak{I}'_{m'} = \langle$R-UP, C, AV$\rangle$

figure 3-12: syntax and semantics of the *insert_attribute* operation

4. **delete attribute**: deletes an existing, but disconnected attribute (i.e., the attribute is not attached to a dimension level or fact).
Preconditions: the attribute exists ($a_{del} \in A$) and must not be connected to a fact or to a dimension level ($attr(a_{del}) = \perp$).

   Postcondition: after the application of the operation, the attribute is not contained in the MD model.

   Syntax and semantics:

delete_attribute	
syntax with input and output parameters	delete_attribute ($\mathcal{M}, \mathfrak{I}_m, a_{del}$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, attribute name a_{del} to be deleted **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}' = \langle F, L, A', gran, class, attr' \rangle$ $A' = A - \{ a_{del} \}$ $attr': A' \rightarrow F \cup L \cup \{\perp\}$; $attr'(a) := attr(a)\ \forall\ a \in A'$ **Instances:** no effect, thus: $\mathfrak{I}'_{m'} = \langle$R-UP, C, AV$\rangle$

figure 3-13: syntax and semantics of the *delete_attribute* operation

5. **connect attribute to dimension level**: connects an existing attribute a_{new} to an existing dimension level *l*. A function *g* assigns values (default or computed) for the new attribute to every member (instance) of the dimension level.

   Preconditions: the attribute and the dimension level exist ($a_{new} \in$ A, $l \in$ L), the attribute must not be connected to another element (attr(a_{new})=⊥). Further, *g* must be well-defined for all dimension members of the level: g(m)=v with v∈ dom(a_{new}) ∀m ∈ dom(l).

   Postconditions: after the application of the operation, the attribute is not isolated anymore. All dimension members have values for the new attribute. The resulting MD model is consistent.

   Syntax and semantics:

connect_attribute_to_dim_level	
syntax with input and output parameters	connect_attribute_to_dim_level ($\mathcal{M}$, $\mathfrak{I}_{\mathcal{M}}$, a_{new}, l, g) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_{\mathcal{M}}$, attribute a_{new} to be connected, dimension level l to which a_{new} is connected, function g for the computation of the a_{new} values **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{\mathcal{M}'}$
semantics	**Schema:** $\mathcal{M}'$=<F, L, A , *gran*, *class*, *attr'*> $attr' : A \rightarrow F \cup L \cup \{\perp\}$ $\quad attr'(a) := \begin{cases} l & if\ a = a_{new} \\ attr(a) & if\ a \neq a_{new} \end{cases}$ **Instances:** $\mathfrak{I}'_{\mathcal{M}'}$ = <R-UP, C, AV'>, AV': AV' := AV ∪ {av$_{anew}$}, define av$_{anew}$: dom (l) → dom(a$_{new}$) with $av_{a\,new}(m) := g(m)\ \ \forall m \in dom(l)$

figure 3-14: syntax and semantics of the *connect_attribute_to_dim_level* operation

6. **disconnect attribute from dimension level**: disconnects an attribute a_{del} from a dimension level $l \in$ L. The operation merely removes the *is_attribute_of relationship*, leaving both the attribute and the dimension level as parts of the MD model.

   Preconditions: the attribute and the dimension level exist ($a_{del} \in$A, $l \in$ L). Further, the attribute and dimension level must be connected to each other ($attr(a_{del})=l$).

   Postconditions: after the application of the operation, both the attribute and the dimension level still exist, but not connected to each other anymore (both may still be connected to other MD model elements). In particular, the dimension members still exist.

Syntax and semantics:

disconnect_attribute_from_dim_level	
syntax with input and output parameters	disconnect_attribute_from_dim_level ($\mathcal{M}, \mathfrak{I}_m, a_{del}, l$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, attribute a_{del} and level name *l* to be disconnected **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}'$=<*F, L, A, gran, class,* attr'> $attr' : A \rightarrow F \cup L \cup \{\perp\}$ $\quad attr'(a) := \begin{cases} \perp & if\ a = a_{del} \\ attr(a) & if\ a \neq a_{del} \end{cases}$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP, C, AV'>, AV': AV' := AV - { av_{adel} }with av_{adel} being the corresponding attribute value function for a_{del}

figure 3-15: syntax and semantics of the *disconnect_attribute_from_dim_level* operation

7. **connect attribute to fact:** connects an existing attribute a_{new} to an existing fact *f*. A function *g* assigns values (default or computed) for the new attribute to every instance of the fact.

   Preconditions: the attribute and the fact exist ($a_{new} \in$ A, $f \in$ F), the attribute must not be connected to another element (attr(a_{new})=$\perp$). Further, *g* must be well-defined for all fact instances.

   Postconditions: after the application of the operation, the attribute is not isolated anymore. All fact instances have values for the new attribute. The resulting MD model is consistent.

   Syntax and semantics:

connect_attribute_to_fact	
syntax with input and output parameters	connect_attribute_to_fact ($\mathcal{M}$, $\mathfrak{I}_m$, a_{new}, f, g) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, attribute a_{new} to be connected, fact *f* to which a_{new} is connected, function *g* for the computation of the a_{new} values **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}'$=<F, L, A , *gran, class, attr'*> $attr' : A \rightarrow F \cup L \cup \{\perp\}$ $\quad attr'(a) := \begin{cases} f & if\ a = a_{new} \\ attr(a) & if\ a \neq a_{new} \end{cases}$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP, C', AV>,

	C' := C – {c_f} ∪ {c_f'} with c_f being the existing cube for f ; define c_f': dom(f) → codom(f) as $c'_f(x) := (z_1,\ldots,z_n,z_{n+1})$ *with* $(z_1,\ldots,z_n) = c_f(x)$ *and* $z_{n+1} = g(x)$

figure 3-16: syntax and semantics of the *connect_attribute_to_fact* operation

8. **disconnect attribute from fact:** disconnects an attribute a_{del} from a fact $f \in$ F. The operation merely removes the *is_attribute_of relationship*, leaving both the attribute and the fact as parts of the MD model.

   Preconditions: the attribute and the fact exist ($a_{del} \in$A, $f \in$ F). Further, the attribute and fact must be connected to each other ($attr(a_{del})=f$).

   Postconditions: after the application of the operation, both the attribute and the fact still exist, but not connected to each other anymore (both may still be connected to other MD model elements).

   Syntax and semantics:

disconnect_attribute_from_fact	
syntax with input and output parameters	disconnect_attribute_from_fact ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}, a_{del}, f$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_{\mathcal{M}}$, attribute a_{del} and fact f to be disconnected **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{\mathcal{M}'}$
semantics	**Schema:** $\mathcal{M}'$=<*F, L, A, gran, class*, attr'> $attr' : A \rightarrow F \cup L \cup \{\perp\}$ $\quad attr'(a) := \begin{cases} \perp & \text{if } a = a_{del} \\ attr(a) & \text{if } a \neq a_{del} \end{cases}$ **Instances:** $\mathfrak{I}'_{\mathcal{M}'}$ = <R-UP, C', AV>, C' := C – {c_f} ∪ {c_f'} with c_f being the existing cube for f ; define c_f': dom(f) → codom(f) as $c'_f(x) := (z_1,\ldots,z_{n-1})$ *with* $(z_1,\ldots,z_{n-1},z_n) = c_f(x)$

figure 3-17: syntax and semantics of the *disconnect_attribute_from_fact* operation

3.7.3. Modification of a classification relationship

9. **insert classification relationship**: this operations defines a classification relationship between two existing dimension levels. The dimension levels may be either isolated elements of the MD model or already be connected by other relationships. If one or both dimension

levels do not contain instances yet (because they are isolated elements), the corresponding classification relationship for the instances has to be defined.

Preconditions: both dimension levels must exist ($l_1 \in L$, $l_2 \in L$) and must not be connected by an existing classification relationship (i.e. $\{(l_1,l_2)\} \notin$ class and $\{(l_2,l_1)\} \notin$ class). The classification relationship between the instances must be well-defined.

Postconditions: after the application of the operation, both dimension levels are connected to each other. A classification between their instances is defined. If one or both of the dimension levels have been isolated elements before, the resulting MD model is consistent after the operation execution.

Syntax and semantics:

insert_classification	
syntax with input and output parameters	insert_classification ($\mathcal{M}, \mathfrak{I}_m, l_1, l_2$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, two dimension level names l_1, l_2 to be connected. **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}' = <F, L, A, gran, class', attr>$ $class' = class \cup \{(l_1,l_2)\}$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP', C, AV>, R-UP' := R-UP $\cup$ { $r-up_{l1}^{l2}$ }, $\forall$ m $\in$ dom(l_1): $r-up_{l1}^{l2}$ (m):= k with k$\in$ dom(l_2). Additionally, $r-up_{l1}^{l2}$ (dom(l_1))$\subseteq$ dom(l_2), i.e., $r-up_{l1}^{l2}$ is well-defined $\forall$ m $\in$ dom(l_1).

figure 3-18: syntax and semantics of the *insert_classification* operation

10. **delete classification relationship**: removes an existing classification relationship between two dimension levels without deleting the corresponding dimension levels. After this operation, the dimension levels may be isolated elements. In particular, the classification information between the instances of the two dimension levels is lost.

    Preconditions: both dimension levels exist ($l_1 \in L$, $l_2 \in L$) and are connected by a classification relationship (i.e. $\{(l_1,l_2)\} \in$ class).

    Postconditions: after the application of the operation, both dimension levels are disconnected from each other. One or both dimension levels may be isolated elements after the operation execution.

Syntax and semantics:

delete_classification	
syntax with input and output parameters	delete_classification ($\mathcal{M}, \mathfrak{I}_m, l_1, l_2$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, two dimension level names l_1, l_2 to be disconnected. **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}'$=<*F, L, A, gran, class', attr*> *class'* = *class* – {(l_1,l_2)} **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP', C, AV>, R-UP' := R-UP - { $r - up_{l1}^{l2}$ }

figure 3-19: syntax and semantics of the *delete_classification* operation

3.7.4. Modification of a fact

11. insert fact: this operation extends an existing MD model by a new fact. The operation extends the set of facts without attaching dimension levels to this fact, thus creating an isolated fact element. Dimensions for this fact have to be defined separately.

Precondition: the new fact may not be contained in the MD model yet ($f_{new} \notin F$).

Postconditions: after the application of the operation, the new fact is part of the MD model. The new fact has no instances.

Syntax and semantics:

insert_fact	
syntax with input and output parameters	insert_fact ($\mathcal{M}, \mathfrak{I}_m, f_{new}$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, fact name f_{new} to be inserted **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}'$=<*F', L, A, gran', class, attr'*> *F' := F* $\cup${f_{new}}, $gran'\colon F' \to 2^L$, $gran'(f) := \begin{cases} \varnothing & if \quad f = f_{new} \\ gran(f) & if \quad f \neq f_{new} \end{cases}$ *attr' := attr* **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP, C', AV>, *C'*:= $C \cup \{ c_{f\,new} \}$,

	define $c_{f\,new}$: dom (f_{new}) → codom(f_{new}) as c(x):= ⊥ ∀ x∈ dom(f_{new})

figure 3-20: syntax and semantics of the *insert_fact* operation

12. delete fact: removes an existing, but isolated (i.e. not connected to any other element of the MD model) fact f_{del} from an MD model. Instances are deleted automatically.

Preconditions: the fact exists ($f_{del} \in F$). The fact must not be connected to a dimension ($gran(f_{del})=\emptyset$) and must also not contain any attributes ($attr(a) \neq f_{del} \forall a \in A$).

Postconditions: after the application of the operation, the fact is not contained in the MD model. Existing instances are deleted.

Syntax and semantics:

delete_fact	
syntax with input and output parameters	delete_fact ($\mathcal{M}, \mathfrak{I}_m, f_{del}$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, fact name f_{del} to be deleted **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}' = <F', L, A, gran\|_{F'}, class, attr' >$ $F' := F - \{f_{del}\}$, $attr' := attr$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP, C', AV>, $C' := C - \{ c_{f\,del} \}$

figure 3-21: syntax and semantics of the *delete_fact* operation

13. insert dimension level into fact: inserts an existing dimension at the specified dimension level into an existing fact (relationship), thus increasing the number of dimensions by one. Parameters are the level name and the fact name that are to be connected.

Additionally, a function *nv* has to be provided defining how the new values for the fact can be computed based upon the now extended set of dimensions and the old value of the fact. Each cell of the old cube now becomes a set of cells, exactly reflecting the new dimension. This means that each old value of the fact is now related to all elements of the new dimension. For instance, we assume daily repair cases of cars stored without the brand (i.e., we have no distinction between the brand of cars). Now we want to include the brand meaning that we insert a new dimension at the level brand (see figure 3-22).

To this end, we have to provide a function that computes the new fact (repair cases by brand) based on the old dimensions (without brand) and the (old) number of repair cases.

The old number of repair cases could be repair cases for a specific brand (alternative 1 in figure 3-22), a summarization over all brands (alternative 2), or other. The idea how the new values can be computed is stored in the function *nv*. For example, if we only had BMW cars before, then we would use the old fact value for BMW and "⊥" for all other cars (because the values cannot be computed, alternative 1). If the old value was a sum over all brands, we could only take this value as a sum, whereas values for the single brands are unknown (corresponding to "?" in figure 3-22).

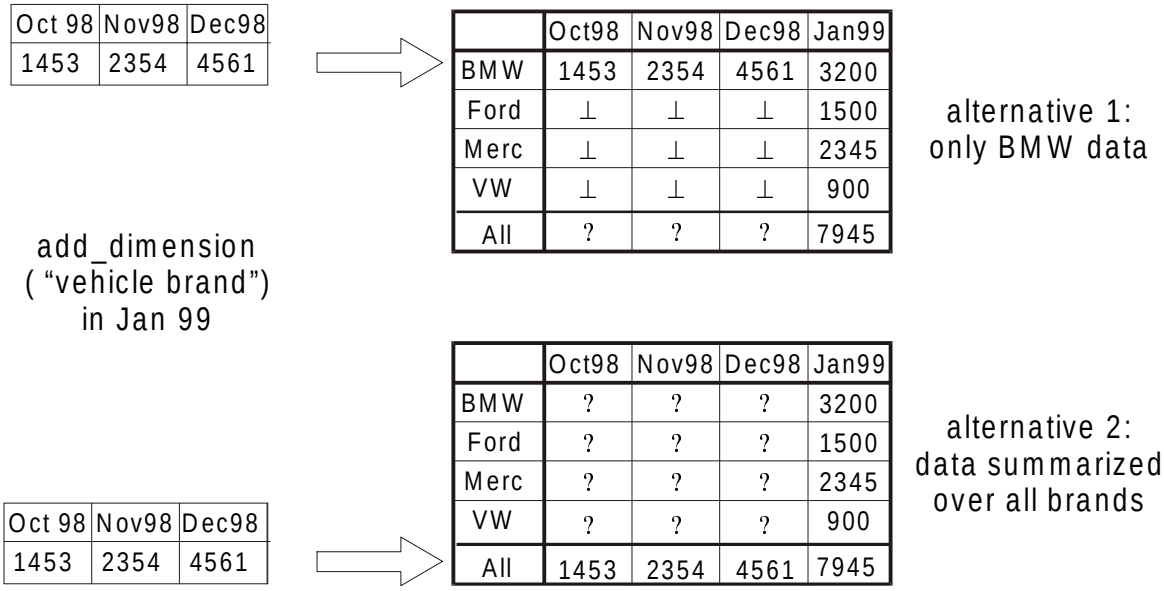


figure 3-22: different alternatives for the instance adaptation

Preconditions: both the fact and the dimension level must exist ($l \in L$, $f_{ins} \in F$) and may not be connected in the MD model yet ($\{l\} \not\subset gran(f_{ins})$). The function *nv* must be well-defined for all existing fact instances.

Postconditions: after the application of the operation, the fact has been extended by one more dimension. The existing fact instances have been adapted w.r.t the new dimension according to function *nv*.

Syntax and semantics:

insert_dimension_into_fact	
syntax with input and output parameters	insert_dimension_into_fact ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}, l, f_{ins}, nv$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_{\mathcal{M}}$, level name *l* and fact name f_{ins} to be connected. Function *nv* to compute the distribution of existing fact instances over the new dimension. **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{\mathcal{M}'}$
semantics	**Schema:** $\mathcal{M}' = < F, L, A, gran', class, attr >$

	define gran': $F \rightarrow 2^L$ *as* $gran'(f) := \begin{cases} gran(f) & for \quad f \neq f_{ins} \\ gran(f) \cup \{l\} & \quad\quad f = f_{ins} \end{cases}$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP, C', AV> C' := C – {c_f} ∪ {c_f'} with c_f denoting the existing cube for f_{ins}. Although the fact f_{ins} itself does not change, its domain changes and the values of its co-domain have to be adapted. Consequently, we define a new cube c_f' and speak of f (or dom(f), codom(f)) if we refer to c_f and speak of f ' (or dom(f '), codom(f ')) if we refer to c_f'. We assume a dimensionality of *n* for c_f and a dimensionality of *n+1* for c_f'. c_f' is derived from c_f as follows: first, we compute the instances of dom(f '): for every combination (x_1,...,x_n,x_{n+1}) ∈ dom(f) in c_f, add | dom(l) | new cube cells (x_1,...,x_n,x_{n+1}, y) with y ∈ dom(l) to c_f'. Second, compute the instances of codom(f '), i.e. adapt the measures: $c'_f(x_1,\ldots,x_n,x_{n+1}) = nv(c_f(x_1,\ldots,x_n),x_{n+1})$ with $nv : codom(f)\ x\ dom(l) \rightarrow codom(f')$ being the function that distributes the existing measures over the new dimension.

figure 3-23: syntax and semantics of the *insert_dimension_into_fact* operation

14. delete dimension level from fact: deletes a dimension, specified by the dimension level, from a fact. The operation disconnects the base level *l* for this dimension from the fact f_{del}. Neither the fact nor the dimension level are deleted implicitly. Since the dimensionality of the fact is reduced, an aggregation function *agg* has to be provided which defines how the existing measures are aggregated over the deleted dimension (e.g. by summation).

Preconditions: both the fact and the dimension level must exist (l ∈ L, f_{del} ∈ F) and must be connected to each other ({l} ⊆ gran(f_{del})). The function *agg* must be well-defined for all existing fact instances.

Postconditions: after the application of the operation, the dimensionality of the fact has been reduced by one dimension, possibly leaving a zero-dimensional fact. The existing fact instances have been aggregated w.r.t the function *agg*.

Syntax and semantics:

delete_dimension	
syntax with input and output parameters	delete_dimension ($\mathcal{M}, \mathfrak{I}_m, l, f_{del}, agg$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, level name *l* and fact name f_{del} to be disconnected. Function *agg* to aggregate the existing fact instances over the deleted dimension. **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
semantics	**Schema:** $\mathcal{M}' = < F, L, A, gran', class, attr >$ *define gran'*: $F \rightarrow 2^L$ *as* $gran'(f) := \begin{cases} gran(f) & for \quad f \neq f_{del} \\ gran(f) - \{l\} & \quad\quad f = f_{del} \end{cases}$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP, C', AV> C' := C – {c_f} ∪ {c_f'} with c_f denoting the existing cube for f_{del}. Although the fact f_{del} itself does not change, its domain changes and the values of its co-domain have to be adapted. Consequently, we again define a new cube c_f' and speak of f (or dom(f), codom(f)) if we refer to c_f and speak of f ' (or dom(f '), codom(f ')) if we refer to c_f'. We assume a dimensionality of *n* for c_f and a dimensionality of *n-1* for c_f'. We further assume that the dimension to be deleted corresponds to the *n*-th element in *dom(f)*. c_f' is derived from c_f as follows: c_f' *: dom (f ') → codom (f ') with dom(f ') being the reduced domain and* $c'_f(x_1,\ldots,x_{n-1}) = agg_{x_n}(c_f(x_1,\ldots,x_n)) \quad with\ x_n \in dom(l)$

figure 3-24: syntax and semantics of the *delete_dimension* operation

So far, we have introduced a set of 14 atomic schema evolution operations. As already said, their main benefit is the proximity to the ME/R graph representation of a given MD schema. But, as a drawback, the execution of such an operation may corrupt the consistency of a given MD schema e.g. by creating isolated elements that are not connected to any other MD schema element. Since a typical schema design session delivers a sequence of schema evolution operations, we now close this gap by showing where and how we check consistency in this schema design and evolution process. We also define some properties that a sequence of evolution operations ideally should fulfill to allow for efficient processing in the target system on the logical layer.

3.8. Evolution Operation Sequences and Consistency

According to our formal approach defined in chapter 3.4 and having our vision of schema design with a graphical modeling tool in mind, we now present where and how we apply our formal results in the overall user interaction scenario.

First of all, we note that the ME/R model, visualized as a typed graph constitutes the interface to the user. There, he performs graph operations like adding or deleting nodes and/or edges. Since the graph corresponds to the MD schema, he actually performs a sequence of schema evolution operations, defined on the graphical representation of the MD schema. We assume without loss of generality that he starts his schema design session with a consistent MD schema (or a correct ME/R graph, accordingly). His graph operations correspond to a sequence of schema evolution operations. Finally, when the modified conceptual MD schema fits the requirements of the changed universe of discourse, he finishes his schema design session.

At this point, the resulting MD schema has to be checked for consistency. The check is performed on the ME/R graph (by checking the correctness criteria for ME/R graphs). Then, we use the isomorphism between an ME/R graph and its corresponding MD schema in the way that the correctness of the ME/R graph assures certain properties on the MD schema, namely the consistency of the MD schema. It also guarantees that the sequence of evolution operations has transformed a consistent MD schema to another consistent MD schema. We will see later (in chapter 4) how these properties are exploited for generating and optimizing the DDL/DML commands to adapt the logical schema and instances.

Currently, we assume that the sequence of schema evolution operations is obtained by logging the user's graph operations, each of which corresponds to a schema evolution operation. Of course, when only regarding the MD schema before the schema evolution session and afterwards, there is in general an infinite number of operation sequences transforming an MD schema to another. The approach of recording the user's graph interactions delivers not necessarily the optimal sequence. However, we do not focus on the issue of how to obtain the optimal sequence (nevertheless, we discuss this question in chapter 5), but from the infinite number of possible operation sequences, we mark one operation sequence γ (see chapter 3.4) which is optimal in the following sense:

(1) the sequence has minimal length. Especially, this means that the sequence contains no compensating operations (e.g. an *insert dimension level l* operation, followed by a *delete dimension level l* operation)

(2) the sequence transforms a correct ME/R graph to another correct ME/R graph.

(3) the sequence is ordered in a way that the pre- and post-conditions of the schema evolution operations regarding other edges and nodes are fulfilled. A simple heuristics (nodes must be inserted before edges between these nodes are defined) guarantees the fulfillment of this requirement. The heuristics can be easily applied by re-ordering the operation sequence accordingly.

Chapter 4 shows how this optimal sequence of schema evolution operations can be automatically processed by our tool environment.

We conclude with the formal theorems and proofs that form the fundamental for the rather informal ideas presented in this chapter. First of all, we have to show that our schema evolu-

tion operations are complete, i.e. that we can generate every possible consistent MD schema by applying our operations:

Theorem 3-5: Completeness of the schema evolution operations

Every consistent MD schema $\mathcal{M}$=<F, L, A, gran, class, attr> can be generated by a sequence of schema evolution operations γ.

◊

Proof 3-5: Completeness of the schema evolution operations

Since the proof makes strong use of the isomorphism between MD schemas and ME/R graphs, we omit the full formal proof because it is very similar to the isomorphism proofs in chapter 3.6. Basically, when starting with an empty MD schema (or corresponding empty graph), we can generate the ME/R graph that corresponds to the MD schema by applying our schema evolution operations, expressed by adding corresponding typed nodes and edges. For example, for every $f \in F$ we introduce a node of type *fact* and so on.

q.e.d. ◊

As a corollar, we conclude:

Theorem 3-6: Transformation between consistent MD schemas

There is always a (not necessarily unique) sequence of schema evolution operations $\gamma = (co_1, co_2, \ldots, co_n)$ with each co_i being an operation as defined in chapter 3.7 that transforms a consistent MD schema $\mathcal{M}_1$=<F_1, L_1, A_1, $gran_1$, $class_1$, $attr_1$> to another consistent MD schema $\mathcal{M}_2$=<F_1, L_2, A_2, $gran_2$, $class_2$, $attr_2$>.

◊

Proof 3-6: Transformation between consistent MD schemas

The proof is a consequence of Theorem 3-5 and the isomorphism between ME/R graphs and MD schemas. We denote the invariant part of the MD schema as I:= $\mathcal{M}_1 \cap \mathcal{M}_2$ (defined on the components).

For every element of I (precisely: every element of a component of I) we then construct the sequence of evolution operations as follows:

For every element (i.e. node or edge in the corresponding ME/R graph) contained in $\mathcal{M}_1 \setminus I$, add the corresponding *delete/disconnect* operation.

For every element contained in $\mathcal{M}_2 \setminus I$, add the corresponding *insert/connect* operation.

q.e.d. ◊

This theorem enables our approach presented above. Now we have formally proven that there always exists a sequence of schema evolution operations (γ) that transform the MD schema at the beginning of the evolution session (Σ_C) to the MD schema at the end of the evolution session (Σ_C'). To complete our approach, we have to show that there exists at least one sequence γ which fulfills our criteria of optimality. We omit the formal proof and just sketch the underlying ideas which contribute much more to the understanding of our approach.

The first property is minimality. Minimality in length can be easily achieved by deleting compensating operations from the recorded sequence of operations.

Property two is the transformation of a correct ME/R graph to another correct ME/R graph. The check of the ME/R graph before and after a schema evolution session ensures that we have applied an operation sequence which maintains correctness. Another advantage of checking the correctness of the resulting ME/R model at the end of a schema design session is the fact that incorrect schemas or operation sequences are detected before the schema evolution operations are processed. If inconsistencies would be detected while processing the evolution in the target system, aborted transactions and corresponding rollbacks would decrease the overall system performance.

Finally, the last property is concerned with the correct ordering within the operation sequence w.r.t. the pre- and postconditions of the operations referring to existing nodes and edges. The simple heuristics allows a re-ordering that exactly fulfills these pre- and postconditions. The heuristics can also very easily be implemented in a graphical modeling tool: edges may only be defined between already existing nodes. Especially, this does not restrict the tool capabilities, but reflects the canonical and intuitive understanding of building and modifying graphs in a modeling tool.

3.9. Summary

This chapter presented the formal core of FIESTA. Consequently, we summarize the main contributions here.

After an extended motivation which presented as main contribution a generic roadmap to schema evolution, we showed an example of a schema evolution case in our graphical modeling tool. We think that this example awoke a general, but still rather incomplete, idea and vision of the main scientific contributions of FIESTA and its implementation as part of the BabelFish project. In particular, we sketched a code fragment of the generated DML/DDL commands that perform the schema evolution on the logical OLAP layer, which leads us to chapter 4 where we describe how schema evolution operations are processed in the underlying relational database and OLAP system configuration.

Next, we presented three groups of objectives for FIESTA: objectives concerning the evolution algebra, the execution model and the software architecture. These objectives are a necessary prerequisite for the understanding of FIESTA's vision and main contributions.

After the objectives, we formalized our approach to multidimensional schema evolution. To that end, we introduced a kind of formalized bird's view on the next chapters. We especially pointed out our notions of consistency and the difference between schema evolution on the conceptual and the logical layer, and concluded with a formal definition of the research problem that FIESTA addresses.

Our multidimensional data model presented the first refinement step of the formalized approach to multidimensional schema evolution. We put a special focus both on the informal introduction to the underlying ideas of our formal data model and the main differences to other formalizations that have been proposed in the literature. Our running example helped in understanding the corresponding definitions of MD schemas and MD instances. We also refined our notion of consistency and presented a set of formal integrity constraints for consistent MD schemas.

The next chapter formalized a dualism which actually has been used informally during the whole thesis so far: the dualism of an ME/R model and its algebraic counterpart in terms of our MD data model. To that end, we introduced ME/R graphs as special case of typed graphs. We showed that a normalization step for ME/R graphs is necessary to enable the isomorphism between ME/R graphs and MD schemas and added special conditions guaranteeing correctness for ME/R graphs. We then presented "construction plans" to map a correct ME/R graph to a consistent MD schema and vice versa. All these prerequisites allowed us to define the isomorphism between the graph-oriented view and the algebraic description of a given MD schema.

As next core piece of FIESTA and as main research contribution, we introduced our schema evolution operations. After an explanation why we used a very fine-grained approach, we formally described a set of fourteen schema evolution operations for the multidimensional data model. To facilitate the understanding, we added examples and informal explanations, where necessary.

Finally, the last chapter closed the open bracket of how the evolution operations are used in our graphical schema design and maintenance tool. We showed that a typical schema designer session delivers a sequence of schema evolution operations. Since the user works with the ME/R graph, his interactions are graph modifications which exactly correspond to the schema evolution operations. At the end of his design or maintenance session, he "checks in" his modified schema, visualized by the ME/R graph. We decided to check the correctness of the resulting ME/R graph immediately, because inconsistencies detected during processing of the evolution operations in FIESTA would need corresponding transactions on the logical layer (i.e. in the underlying database) to be rolled back.

Chapter 4 will now describe how the schema evolution operations are transformed to corresponding DML/DDL commands that adapt the logical schema and instances together with the OLAP tool metadata. The main parameter for the next chapter, delivered from the formal core presented here, is a sequence of fine-grained schema evolution operations together with the (consistent) MD schema before and after the evolution session. For processing the schema evolution operations, FIESTA can rely on certain properties of the two MD schema states and the operation sequence. As we will see now, both the fine-grained approach and the properties allow for an automated generation of the corresponding DML/DDL commands that perform the schema evolution on the logical layer. Especially, we present how the fine-grained operations must be grouped together in order to enable the logical schema evolution according to the capabilities and peculiarities of the target OLAP system and database.

My theory of evolution is that Darwin was adopted.

(Steven Wright)

4. Processing MD Schema Evolution Operations in a Relational DBS

Schema evolution operations are grouped to sequences. This chapter describes how the evolution of a conceptual multidimensional schema – specified by a sequence of schema evolution operations – can be processed in an underlying relational database system. According to our layer model defined in chapter 2.2, the database system is responsible for the persistent storage of the OLAP data together with the OLAP tool metadata.

To this end, we first introduce the classical approach to model multidimensional OLAP data in a relational schema, the so-called star schema. As we will see, some semantics of the multidimensional data model get lost when transforming an MD schema to such a relational structure. The metadata represents exactly this information. Consequently, we introduce a meta schema as an extension of the relational database system catalogue to store this information. Basically, the meta schema consists of three parts: one part covers metadata to describe multidimensional schemas. The next part is the corresponding section from the standard RDBMS system catalogue to describe relational tables with their columns. Finally, we need metadata to describe the mapping between the conceptual (multidimensional) and logical (relational) layer. This is done by defining correspondences between elements on both layers.

We then use this mapping between the conceptual and the logical layer to define consistency between the two layers. Basically, this is another view of the interrelationships between the conceptual and logical layer.

We define how the conceptual schema evolution operations are transformed to logical evolution operations. These logical evolution operations adapt the structure of the star schema together with the data (instances) stored in it and update the metadata (stored in the meta schema) accordingly. We will present transformations for sequences of evolution operations (which then form complex operations) to specific logical evolution operations. In order to show the correctness of the evolution, we use the consistency definition to check if the transformations on both layers have according semantics. Finally, some further ideas for optimizing evolution operation sequences will be given.

4.1. Mapping MD Schemas to Relational Database Schemas

As already introduced in chapter 2.2, the logical layer is responsible for the persistent storage of the data in terms of relations or multidimensional arrays[6]. This task is done by a database system. Thus, the logical layer comprises the database schema together with the set of instances representing the actual OLAP data, but also metadata. This metadata represents detailed information about the mapping from the conceptual multidimensional layer to the logical database system layer.

4.1.1. The Relational Database Schema

For OLAP systems, there exist two classical alternatives for the underlying database system: the first is to choose a relational database system (so-called ROLAP architecture) for storage of the multidimensional data, the second is the use of dedicated multidimensional database systems (so-called MOLAP architecture) which use specialized array structures for the persistent storage of the multidimensional data. Since the ROLAP architecture has proven to be more scalable and because relational database systems offer at least basic support for schema evolution - as opposed to no support in multidimensional database systems, we decided to base FIESTA on a ROLAP architecture.

The classical approach for a relational database schema to represent a multidimensional schema, is the so-called star schema ([Kim96a], [Inm96], [Sir97], [McG96]). A star schema organizes each fact (which represents the subject of the analysis) in a relational table, called the fact table. For each dimension, the complete dimension hierarchy (i.e. all dimension levels with their describing attributes) are combined in a relational dimension table. The relationship between the fact table and the dimension tables is maintained by foreign key relationships. This means that the fact table has a combined key, composed of the set of all foreign keys of the dimensions. A star schema template is depicted in figure 4-1:

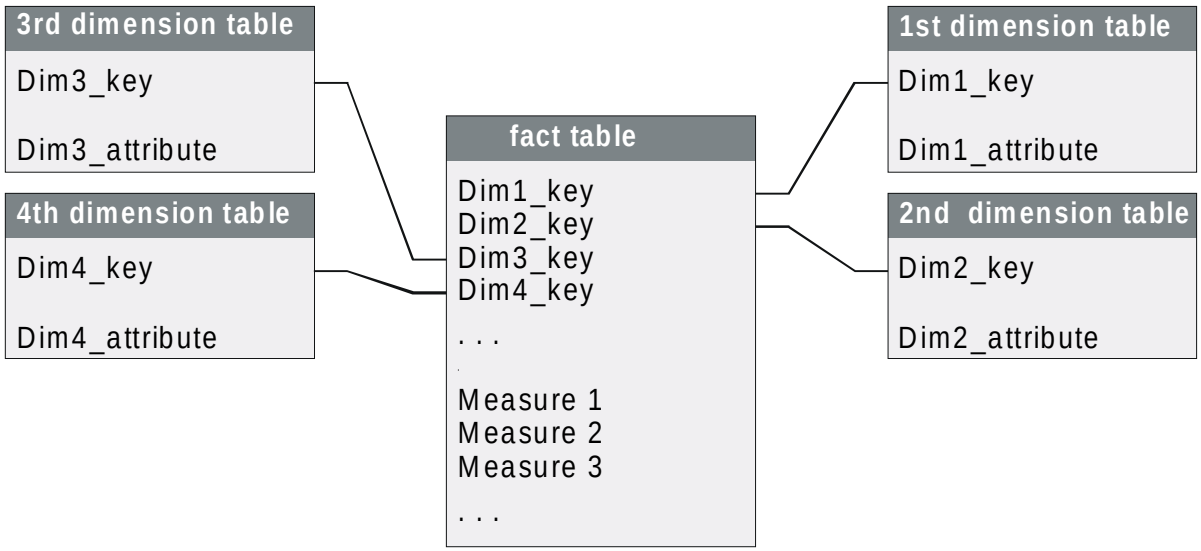


figure 4-1: star schema

[6] We remark that the physical organization of these relations or MD arrays, i.e. clustering techniques for disk storage or indexing strategies, belongs to the physical layer.

As can be easily seen, some semantics of the multidimensional data model are lost or at least hidden in such a star schema. For example, there is no information representing the classification hierarchy of a dimension (because the attributes of a dimension table have no inherent order) or the distinction between dimension levels and describing attributes of a dimension level (because they are both merely attributes in the relational table).

Nevertheless, OLAP tools need this information for mapping user queries which are specified in terms of the multidimensional schema to queries for the underlying database system.

To solve this shortcoming of a star schema, this information is additionally stored as metadata.

As a consequence, we present a meta schema as extension of a standard relational database system catalogue. The meta schema comprises information about the conceptual multidimensional schema as well as information concerning the mapping from multidimensional schemas to relational schemas.

4.1.2. A Meta Schema for MD Schemas

The meta schema part which represents the multidimensional schema is to some degree dependent on the underlying MD data model[7]. Thus, the meta schema part describing MD schemas contains entities that represent facts with measures, dimension levels, the classification hierarchies of the dimensions and the dimensional mapping that assigns the different dimensions to a fact (an E/R diagram of this part of the meta schema is shown in figure 4-2).

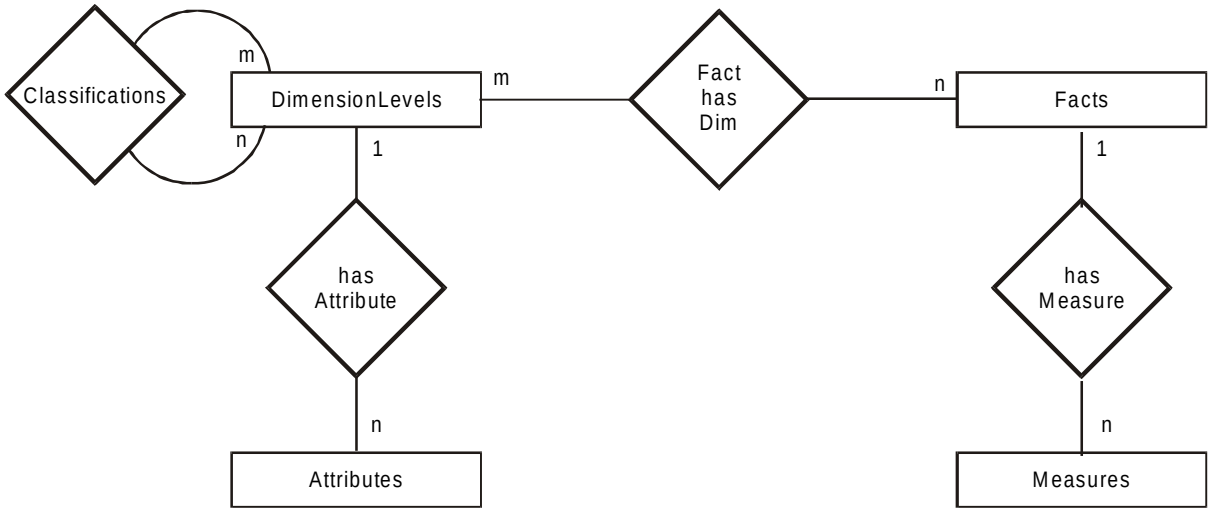


figure 4-2: meta schema for MD schemas

This part of the meta schema represents our basic understanding of the FIESTA multidimensional data model, as defined in chapter 3.5: an MD schema consists of multidimensional facts with measures, the dimensions in turn consist of dimension levels with classification hierarchies. Each dimension level may have describing attributes. More specifically, the entity *Facts* represents information about facts (fact nodes in the ME/R diagram). For each fact, there may be an arbitrary number of *Measures* defined (relationship *hasMeasure*). Since we explicitly allow for shared dimensions and multiple facts in a given model (see chapter 2.4.1), we have a

[7] As a consequence, we note that in commercial tools this part of the meta schema varies slightly [Ulb99], [DSVH97].

n:m relationship *FacthasDim* between *Facts* and *DimensionLevels*. The classification hierarchy of dimension levels is represented by the relationship *Classifications*. Every dimension level may have n (describing) *Attributes* (relationship *hasAttribute*).

We remark that some integrity constraints for an MD schema are not expressible by the mere structure of the meta schema (e.g. prohibition of cycles in the classification hierarchy). In order to check for consistency of an MD schema, additional predicates over the contents of the meta schema would have to be defined and evaluated.

4.1.3. Adding the Relational Meta Schema

The part of the meta schema representing the relational database schema is covered by the standard database system catalogue. The relevant section of this system catalogue consists of relational *Tables* which are composed of *Columns*[8].

The two different parts of the meta schema for both the conceptual multidimensional layer and the logical relational layer are depicted in figure 4-3.

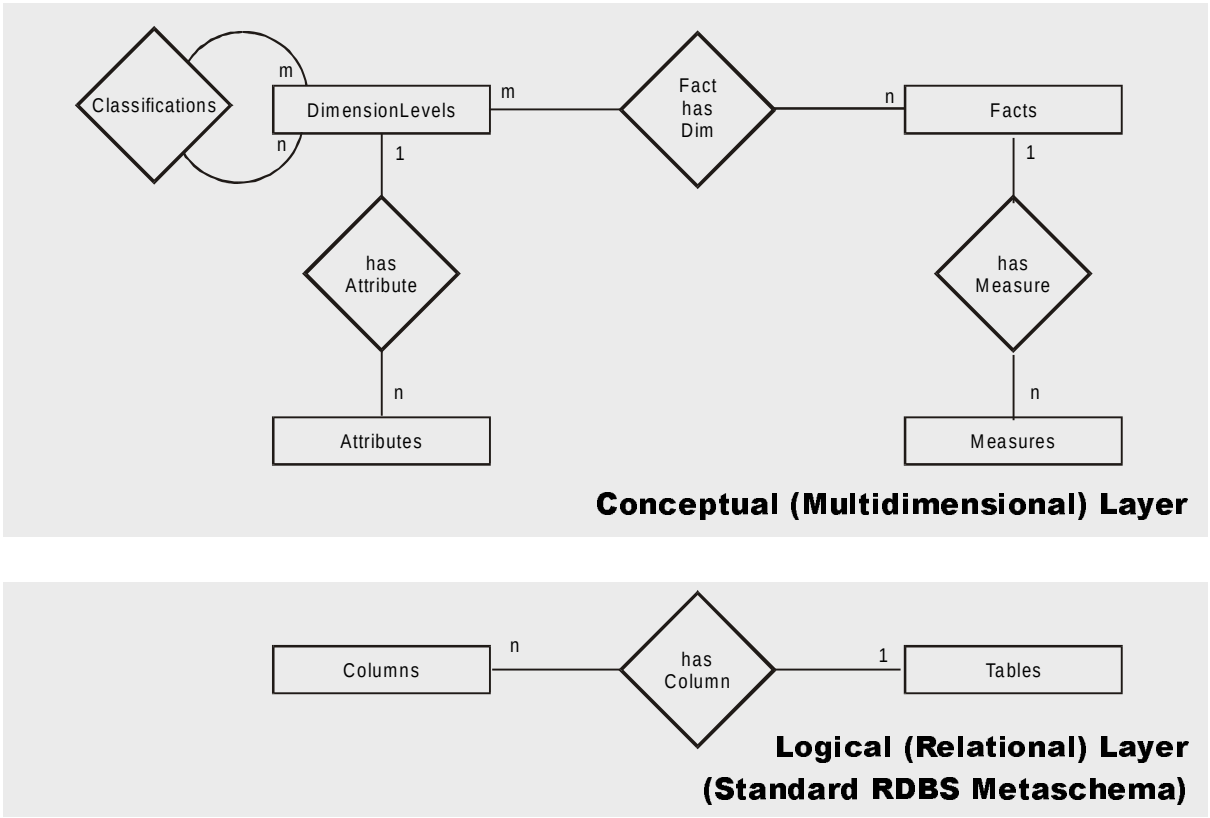


figure 4-3: meta schema for MD schema and star schema

[8] We use the terms tables and columns instead of relations and attributes to avoid confusion with other parts of the meta schema.

4.1.4. Adding the Mapping Information

For the processing of schema evolution operations, we need the correspondences between the MD schema elements and the tables and columns on the relational layer (corresponding to the mapping function α of figure 3-5 in chapter 3.4). Informally, this mapping seems rather straightforward when regarding the classical star schema template (figure 4-1): each fact node becomes a fact table with foreign key relationships to its dimension tables (represented by the base levels of a dimension in the MD schema) and all its measure attributes. Each base level of a fact is transformed to a dimension table, which attributes consist of all levels along the classification hierarchy together with all describing attributes of these levels.

Thus, we need the following mapping correspondances (see also figure 4-4 for the corresponding grey shaded relationship names):

- from facts to fact tables (relationship *FactTableMapping*)
- from measures to columns in the corresponding fact table (relationship *MeasureMapping*)
- from base dimension levels (i.e. those $l \in L$ with $\exists f \in F$ and $l \in gran(f)$) of a fact to their corresponding dimension tables (relationship *DimTableMapping*). Although this information is redundant (each level is mapped to at least one column of a corresponding dimension table, see relations *DimHierarchyMapping* and *hasColumn*), it assists in transforming conceptual schema evolution operations to corresponding logical evolution operations (see chapter 4.4).
- from base dimension levels of a fact to the corresponding (foreign key) columns in the fact table (relationship *FactDimsMapping*)
- from all dimension levels in a classification hierarchy to columns of the corresponding dimension table (relationship *DimHierarchyMapping*)[9]
- from describing attributes to columns in the corresponding dimension table (relationship *AttributeMapping*)

The resulting overall meta schema including the mapping correspondences (grey shaded) is depicted in figure 4-4:

[9] We remark that another integrity constraint is not expressed here: the column representing dimension level l must be part of the proper table, i.e. it must belong to a dimension table (see our naming conventions) and additionally to the corresponding dimension table.

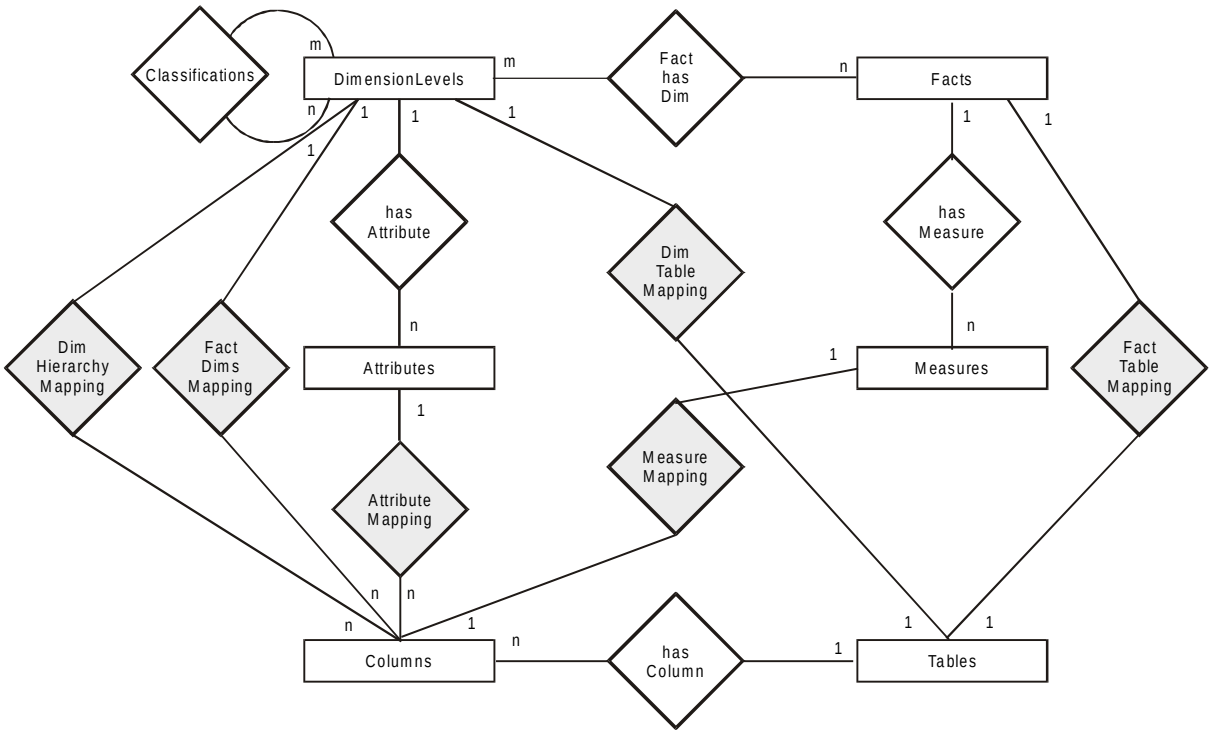


figure 4-4: meta schema with mapping between MD schema and star schema

4.1.5. The complete Meta Schema

The complete meta schema (presented by its relations and attributes) together with the corresponding ME/R elements or MD schema elements (where applicable) is shown in the following figure 4-5.

We omitted surrogate keys (except for the relation *Columns*) which would be used as key attributes in a real implementation. Since determining unique keys is an implementation concept and not necessary for explaining the idea of the meta schema, we omit the surrogate keys here and use the name attribute as keys instead. The only exception to that rule is the relation *Columns* where we need an identifier (ID) as key attribute for explaining the concepts of the meta schema. Column names in relational tables must not be unique and as we will see below, in a typical star schema, the same column names may be defined in different tables. To make clear which column is meant, we decided to use the identifier (ID) as key attribute.

Relation Name	**Attributes**	**Corresponding ME/R Element**	**Corresponding MD Schema Element**
Facts	name: string table_name: string (FK)	fact node	$f \in F$
Measures	name: string fact: string (FK) domain: string column_ID: integer (FK)	attribute connected to fact	$a \in A$ with attr(a)= f for $f \in F$

Dimension Levels	name: string is_base: bool table_name: string domain: string	dimension level	$l \in L$
FacthasDim	fact: string (FK) dim_level: string (FK)	dimension edge	gran(f)
Classifications	dim_level1: string (FK) dim_level2: string (FK)	classification edge	class
Attributes	name: string dim_level: string (FK) domain: string	attribute connected to dimension level	$a \in A$ with attr(a)= l for $l \in L$
FactDimsMapping	dim_level: string (FK) column_ID: integer (FK)	n.a. (mapping of a base level to a foreign key attribute in a fact table)	
DimHierarchy Mapping	dim_level: string (FK) column_ID: integer (FK)	n.a. (mapping of dimension levels to corresponding columns in dimension tables)	
AttributeMapping	attribute: string (FK) column_ID: integer (FK)	n.a. (mapping of attributes to corresponding columns in dimension tables)	
Tables	name: string	n.a. (standard DB system catalogue)	
Columns	ID: integer name: string table_name: string (FK)	n.a. (standard DB system catalogue)	

figure 4-5: FIESTA meta schema

The table *Facts* represents information about the fact nodes of an ME/R model or the facts of an MD schema, respectively. For a fact, we store its (conceptual) name and (as foreign key to *Tables*) the name of the corresponding fact table in the database (1:1 relationship *FactTableMapping*). Since we regard the relations *Tables* and *Columns* as part of the DBMS system catalogue, we decided to resolve this 1:1 relationship by storing the fact table name in the relation *Facts*.

The table *Measures* represents information about the measure attributes of a fact: the measure name, the fact it belongs to, the domain of the measure, and the reference to the corresponding column in the fact table (1:1 relationship *MeasureMapping*). Again, we decided not to extend the DBMS system catalogue (the structure of which we regard as fixed), but to store the column in our meta schema relations which then constitute an extension (that can be easily implemented) to the standard system catalogue.

The relation *DimensionLevels* records the following information: the name of the level and a boolean value that indicates if the level is a base level of a fact. If the level is a base level (and thus forms the lowest level of a dimension hierarchy) the name of the corresponding dimension table is stored (1:1 relationship *DimTableMapping*). If the level is not a base level, the field *table_name* is left empty (NULL value). Altough this information is redundant, it assists the processing of logical evolution operations (see chapter 4.4): the dimension table name can be directly evaluated from the meta schema only by querying the relation *DimensionLevels*. The dimension table name to which a level belongs could also be reached by the relations *DimHierarchyMapping* (to the corresponding column) and *hasColumn* (to the corresponding dimension table). Finally, the domain of the dimension level is stored.

The relation *FacthasDim* represents the multidimensional fact relationships between the facts and their dimensions (denoted by their base levels). To that end, the fact of the dimensional relationship is stored together with every base dimension level.

The relation *Classifications* manages the classification relationships of the MD schema or ME/R model. For every classification relationship, the two connected levels are stored.

The relation *Attributes* contains information about the describing attributes of dimension levels: the attribute name, the dimension level to which the attribute belongs, and its domain.

The relations *FactDimsMapping, DimHierarchyMapping,* and *AttributeMapping* represent the corresponding relationships in the E/R diagram. Although the cardinality of these relationships is 1:n (from *DimensionLevels* (or *Attributes*, resp.) to *Columns*) and thus would be normally modeled as foreign keys in *Columns*, we decided to provide separate relationships for them, because we did not want to extend the standard database system catalogue to which the relation *Columns* belongs.

FactDimsMapping represents the relationship between base levels of a fact and the corresponding foreign key columns (referencing these base levels) in the fact table. To that end, it contains the base level and the corresponding column of the fact table.

DimHierarchyMapping represents the information which dimension level is mapped to which column in a dimension table. Consequently, *DimHierarchyMapping* stores the dimension level and the corresponding column of each dimension table it belongs to. The relationship between *DimensionLevels* and the *Columns* is of cardinality 1:n, because there are two cases when a dimension level is mapped to more than one column of dimension tables:

- merging dimensions and
- multiple facts with shared dimensions.

We briefly discuss these two cases.

Very often the same classification is needed for levels of different dimensions. A prominent example is the geographical classification, which is useful not only for stores, but also for customers, garages, salespersons etc. In figure 4-6 the dimensions customer and garage share the same classification for the geography. In the ME/R model, this means a classification edge between levels of different dimensions.

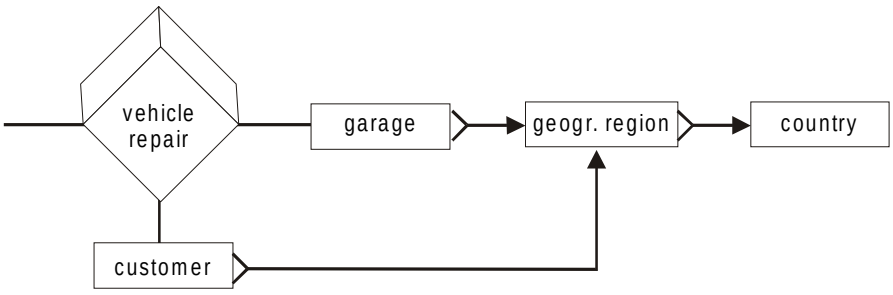
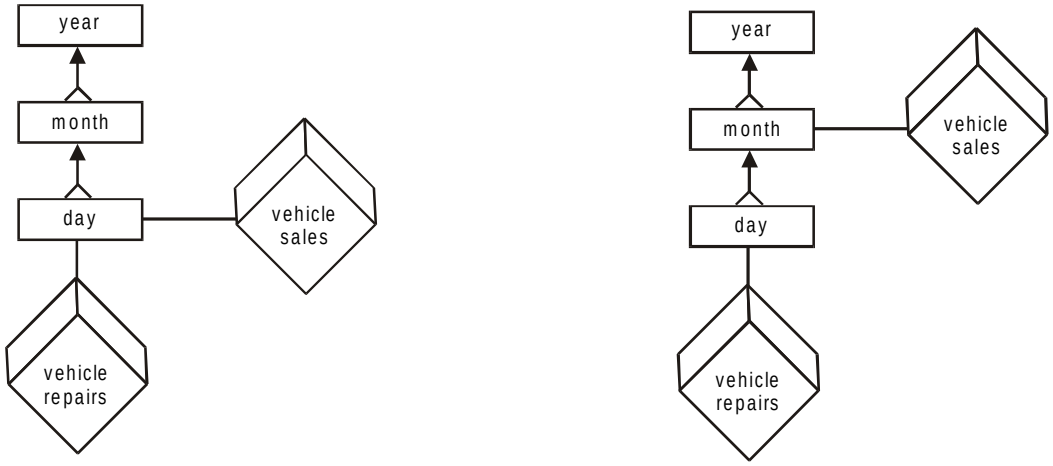


figure 4-6: merging dimensions for geographical classification

This case is called merging dimensions and cannot be directly transformed to a star schema, because dimension tables cannot share their columns. Therefore, when such an ME/R diagram is mapped to a relational star schema, the shared part of the classification hierarchy is duplicated as attributes in the corresponding dimension tables. Consequently, the shared levels of the classification hierarchy are mapped to more than one dimension table columns.

The next case arises when an ME/R model contains more than a single fact and these facts share dimensions. Each of these facts is subject to a different analysis (e.g. vehicle sales and vehicle repairs, see figure 4-7) and has its own measures. Nevertheless, the facts may share dimensions (e.g. the time dimension is contained in almost every fact), either at the same level (left side in figure 4-7: daily repair and daily sales figures) or at different levels (right side in figure 4-7: daily repair, but monthly sales figures) of the dimension hierarchy.



a) shared dimensions at the same level

b) shared dimensions at different levels

figure 4-7: multiple fact nodes with shared dimensions

In the case of shared dimensions at the same level, only one dimension table may exist. Additionally, a dimension level (the shared level) is assigned to more than one fact (which explains the 1:n cardinality of the relationship *FactDimsMapping*).

Shared dimensions at different levels always need separate dimension tables for each base level. As a consequence, the dimension levels above the shared level (which is *month* in the example of figure 4-7 b) are duplicated in the dimension tables (*month* and *year* in the example).

AttributeMapping represents the information which attribute is mapped to which column in a dimension table. Consequently, *AttributeMapping* stores the attribute and the corresponding column of each dimension table it belongs to. The relationship between *Attributes* and the *Columns* is of cardinality 1:n, because the attribute may be assigned to a dimension level that belongs to more than one dimensions.

Finally, the relations *Tables* and *Columns* represent an excerpt from the standard database system catalogue. Consequently, we omit a detailed explanation here. We only repeat that due

to the explanations given above (concerning merging dimensions and multiple facts with shared dimensions), the column names may not be unique for a given star schema. Thus, we introduced the identifier ID as key attribute in *Columns*.

4.2. Example

As an example, we present the relational star schema and meta schema contents for our running example. We repeat the ME/R model for our vehicle repair analysis.

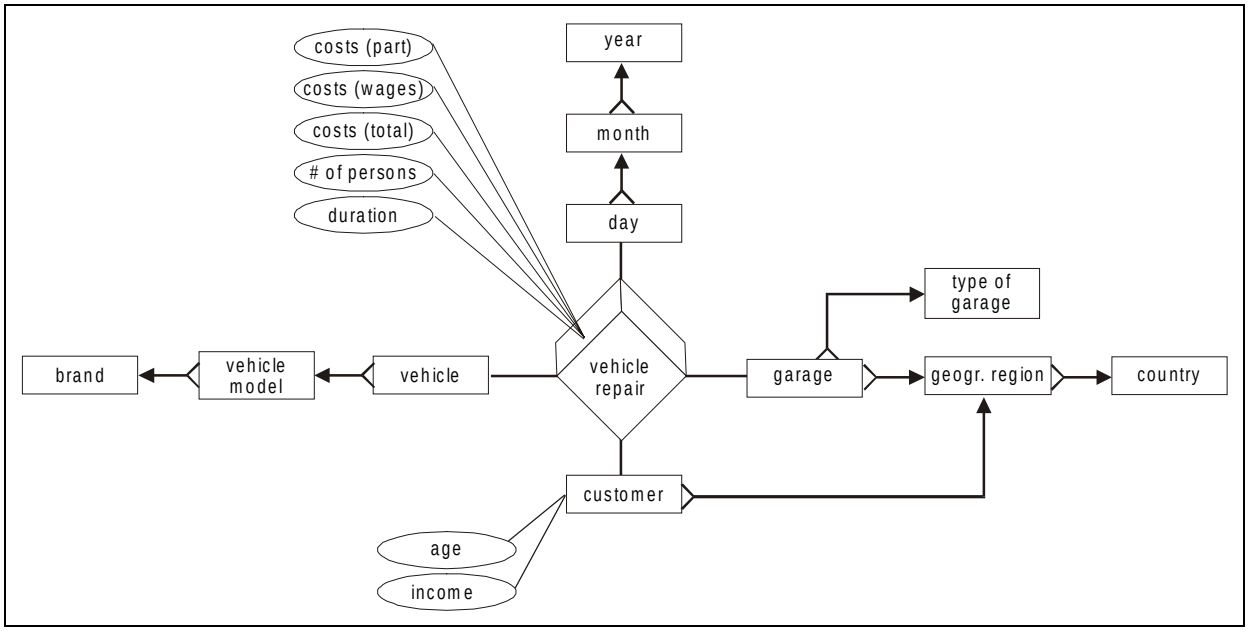


figure 4-8: vehicle repair example

The MD schema $\mathcal{M}_{ex}$=<F_{ex}, L_{ex}, A_{ex}, $gran_{ex}$, $class_{ex}$, $attr_{ex}$> has the following components:
F_{ex} = {vehicle repair}
L_{ex} = {customer, vehicle, vehicle model, brand, day, month, year, garage, type of garage, geogr. region, country }
A_{ex} = {costs (part), costs (wages), costs (total), # of persons, duration, age, income}
$gran_{ex}$(vehicle repair) = {customer, vehicle, day, garage}
$class_{ex}$ = {(day, month), (month, year), (garage, type of garage), (garage, geogr. region), (geogr. region, country), (customer, geogr. region), (vehicle, vehicle model), (vehicle model, brand)}
$attr_{ex}$(“costs (parts)”) = vehicle repair, $attr_{ex}$(“costs (wages)”)= vehicle repair, $attr_{ex}$(“costs (total)”)= vehicle repair, $attr_{ex}$(“# of persons”)= vehicle repair, $attr_{ex}$(duration)= vehicle repair, $attr_{ex}$(age) = customer, $attr_{ex}$(income) = customer

As an anticipation to chapter 4.3, where we use this mapping to define consistency between the conceptual and logical layer and introduce certain naming conventions for the star schema tables, we assume the following dimension tables and fact table:

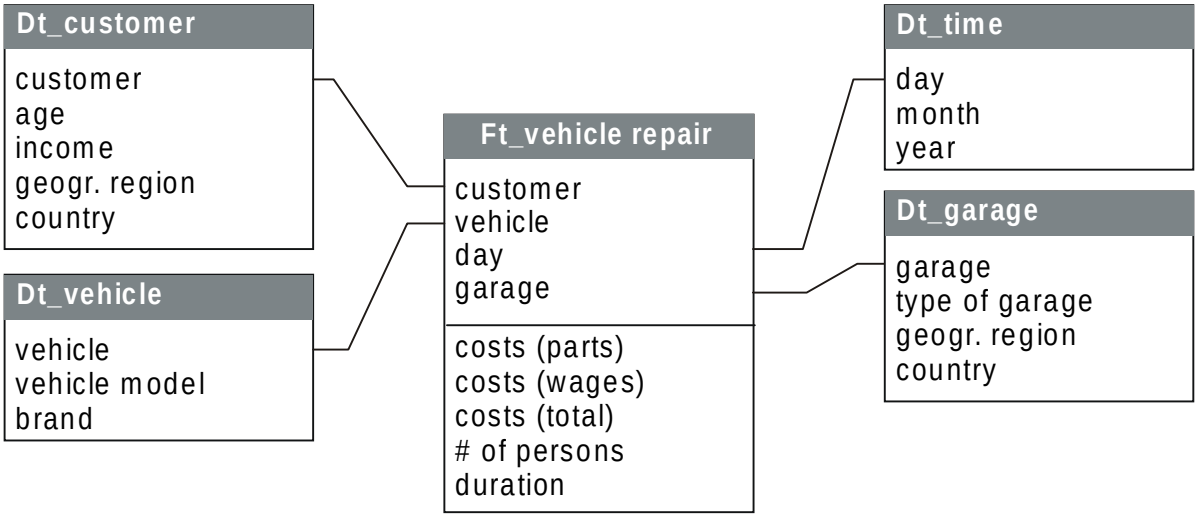


figure 4-9: vehicle repair star schema

We remark that there are some names in the conceptual schema that contain illegal characters (limitations of the relational DBMS) when mapping them to names of relations or attributes. Thus, for a real implementation, we would have to replace those illegal characters (in the example, this means replacing the illegal characters " " (blank) and "." (dot) by underscores and the character "#" by "no_ "). Since this is an implementation concept and not necessary for explaining the idea, we used the names of the conceptual schema also for the relational table and column names.

Further, there is no domain or type information for the attributes in the ME/R model so far. We assume that this information is stored together with the MD schema in the repository. The information may be entered as part of the MD schema design.

In order to complete our example, we present the contents of the meta schema for the vehicle repair example. We start with the relations *Tables* and *Columns*, because the rest of the meta schema is easier to read when already knowing the Column IDs:

Table Name	**Instances**
Tables	(Dt_customer), (Dt_vehicle), (Dt_time), (Dt_garage), (Ft_vehicle repair)
Columns	(1, customer, Dt_customer), (2, age, Dt_customer), (3, income, Dt_customer), (4, geogr. region, Dt_customer), (5, country, Dt_customer), (6, vehicle, Dt_vehicle), (7, vehicle model, Dt_vehicle), (8, brand, Dt_vehicle), (9, day, Dt_time), (10, month, Dt_time), (11, year, Dt_time), (12, garage, Dt_garage), (13, type of garage, Dt_garage), (14, geogr. region, Dt_garage), (15, country, Dt_garage), (16, customer, Ft_vehicle repair), (17, vehicle, Ft_vehicle repair), (18, day, Ft_vehicle repair), (19, garage, Ft_vehicle repair), (20, "costs (parts)", Ft_vehicle repair), (21, "costs (wages)", Ft_vehicle repair),

	(22, “costs (total)”, Ft_vehicle repair), (23, “# of persons”, Ft_vehicle repair), (24, duration, Ft_vehicle repair)
Facts	(vehicle repair, Ft_vehicle repair)
Measures	(“costs (parts)”, vehicle repair, float, 20), (“costs (wages)”, vehicle repair, float, 21), (“costs (total)”, vehicle repair, float, 22), (“# of persons”, vehicle repair, float, 23), (duration, vehicle repair, float, 24)
Dimension Levels	(customer, TRUE, Dt_customer, string), (age, FALSE, NULL, integer), (income, FALSE, NULL, float), (geogr. region, FALSE, NULL, string), (country, FALSE, NULL, string), (vehicle, TRUE, Dt_vehicle, string), (vehicle model, FALSE, NULL, string), (brand, FALSE, NULL, string), (day, TRUE, Dt_time, date), (month, FALSE, NULL, string), (year, FALSE, NULL, string), (garage, TRUE, Dt_garage, string), (type of garage, FALSE, NULL, string)
FacthasDim	(vehicle repair, customer), (vehicle repair, vehicle), (vehicle repair, day), (vehicle repair, garage),
Classifications	(day, month), (month, year), (garage, geogr. region), (garage, type of garage), (geogr. region, country), (customer, geogr. region), (vehicle, vehicle model), (vehicle model, brand)
Attributes	(age, customer, integer), (income, customer, float)
FactDimsMapping	(customer, 16), (vehicle, 17), (day, 18), (garage, 19)
DimHierarchy Mapping	(customer, 1), (geogr. region, 4), (country, 5), (vehicle, 6), (vehicle model, 7), (brand, 8), (day, 9), (month, 10), (year, 11), (garage, 12), (type of garage, 13), (geogr. region, 14), (country, 15)
AttributeMapping	(age,2), (income, 3)

figure 4-10: vehicle repair metadata

4.3. Consistency between the conceptual and logical layer

This chapter formalizes the mapping between an MD schema and a star schema with according contents of the meta schema. The formalization provides a means to check consistency between an MD schema and its relational counterpart, a star schema and corresponding meta schema contents. We need this formal consistency definition later (chapter 4.4) when we show how conceptual schema evolution operations are processed in a relational database.

One might think that this mapping already enables a complete description how to propagate changes in the multidimensional schema to the relational star schema and the contents of the meta schema. However, this is not the case. The mapping defined above works only for the schema transformation (and metadata update). Since a typical OLAP system contains data (instances) when a schema change arises, we also need a means to adapt existing data. This is called instance adaptation. The mapping would only enable to create a new and empty relational schema from a given MD schema. Consequently, we could use this mapping to *generate* relational star schemas (together with corresponding instances of the meta schema) from a given MD schema.

When reflecting again the FIESTA schema evolution scenario (figure 3-5), we see that the starting point is a consistent OLAP system, i.e. a conceptual schema and a corresponding star schema storing the OLAP data, together with according contents of the meta schema. Then the OLAP designer performs changes of the multidimensional schema using his graphical modeling tool. The changes are described by a sequence of schema evolution operations. The question is now how to transform this sequence to SQL DML/DDL commands that adapt the star schema, its instances (which represent the OLAP data) and update the contents of the meta schema. To ensure the correctness of this transformation, we must show that the resulting conceptual schema corresponds to the resulting logical schema and updated contents of the meta schema. Therefore, we need a formal description of this mapping between the conceptual schema and the logical schema together with the contents of the meta schema. This formal description delivers exactly a definition of consistency between the conceptual and logical layer.

We start with some notations and then present the consistency definition.

Definition 4-1: Notations for the consistency between the conceptual and logical layer

We assume an MD schema $\mathcal{M}$=<*F, L, A, gran, class, attr*>.

The dimension tables of a star schema are composed of the set of all levels and corresponding attributes. A dimension corresponds to the set of all dimension levels that are reachable via classification relationships from a given base level of a fact. Thus, we define:

For every $l \in L$ with $\exists\ f \in F$ and $l \in gran(f)$ (i.e. for every base level of a fact):

(1) We denote the set of all levels belonging to a dimension with D_l:

$D_l := \{ m \in L \mid (l,m) \in class^* \}$. The domain of D_l is defined as the cross-product of all domains of levels in D_l.

(2) Similarly, we denote all (describing) attributes of D_l as Attributes (D_l). Formally:

Attributes (D_l) := $\{ a \in A \mid \exists\ n \in D_l$ with attr(a) = n $\}$.

The domain of Attributes (D_l) is defined as the cross-product of all attribute domains.

The fact tables of a star schema are composed of a foreign key to each dimension (i.e. base level) together with the set of the fact's measure attributes. Thus:

For every fact $f \in F$:

(3) We denote the set of all base dimension levels as FK_f. Formally,

$FK_f := \{ l \in L \mid l \in gran(f) \}$. The domain of FK_f is dom(f) (see Definition 3-13)

(4) We denote the set of all measure attributes $Measure_f$ as

$Measure_f := \{ m \in A \mid attr(m) = f \}$. The domain of $Measure_f$ is codom(f).

◊

After having provided the necessary notations, we introduce the consistency between the conceptual and logical layer which extends on the logical layer to both the star schema and the contents of the meta schema.

Definition 4-2: Consistency between MD schema and relational schema with metadata

We assume an MD schema $\mathcal{M}$=<*F, L, A, gran, class, attr*> and a set of relational tables together with a meta schema as defined in chapter 4.1.

The following correspondences between the MD schema and the relational tables together with the metadata must hold in order to ensure consistency:

(1) dimension tables: every dimension level which is directly connected to a fact (we then call the dimension level *base level*) spans a dimension by the classification hierarchy. For every such dimension there must be a separate table. A dimension table has as attributes every dimension level along the classification hierarchy together with all describing attributes of the levels. The dimension table must be registered in the relation *Tables* and all its attributes must be contained in the relation *Columns*. Formally:

for every $l \in L$ with $\exists\ f \in F$ and $l \in gran(f)$:

there is a relation Dim_l with table name "Dt_<l>" and the following attribute set A_{Diml}:

$A_{Diml} := D_l \cup Attributes(D_l)$

with D_l being the set of levels for this dimension and $Attributes(D_l)$ being the attributes of the levels[10].

The relation Dim_l is defined as

$Dim_l \subseteq dom(D_l) \times dom(Attributes(D_l))$

There exists a tuple ("Dt_<l>") in relation *Tables*, for every $a \in A_{Diml}$ there exists a tuple (ID, a, "Dt_<l>") in relation *Columns* with ID being a unique identifier, and for every $m \in D_l$ there exists a tuple (m, cID) in relation *DimHierarchyMapping* with cID referencing the tuple (cID, m, "Dt_<l>") in relation *Columns*. Analogously, for every $a \in Attributes(D_l)$, there exists a tuple (a, colID) in relation *AttributeMapping* with colID referencing the tuple (colID, a, "Dt_<l>") in relation *Columns*.

[10] We remark that in a real implementation of the relational tables, surrogate keys are used instead of the base level elements for the foreign key relationships.

We remark that all other dimension levels that are not base levels of a fact are contained in D_l due to the consistency of the MD schema (see chapter 3.5.4) and the definition of D_l. Similarly, all describing attributes of these levels are contained in Attributes(D_l).

(2) fact tables: a fact table contains as attributes foreign keys of every dimension (defined by its base level) and the measure attributes. The fact table must be contained in relation *Tables* and all its columns in relation *Columns*. Formally:

for every $f \in F$:

there is a relation $Fact_f$ with table name "Ft_<f>" and the following attribute set A_{Ftf}:

$A_{Ftf} := FK_f \cup Measure_f$

with FK_f being the set of base levels from all dimensions and $Measure_f$ being the set of measure attributes of f.

The relation $Fact_f$ is defined as: $Fact_f \subseteq dom(f) \times codom(f)$.

There exists a tuple ("Ft_<f>") in relation *Tables*, for every $a \in A_{Ftf}$ there exists a tuple (ID, a, "Ft_<f>") in relation *Columns* with ID being a unique identifier, and for every $l \in FK_f$ there exists a tuple (l, cID) in relation *FactDimsMapping* with cID referencing the tuple (cID, l, "Ft_<f>") in relation *Columns*.

(3) for every $f \in F$: there exists a tuple (f, "Ft_<f>") in relation *Facts* with "Ft_<f>" being a foreign key in relation *Tables*.

(4) for every $a \in A$ with attr(a) = f for an $f \in F$:
there exists a tuple (a, f, dom(a), cID) in relation *Measures* with f being a foreign key in relation *Facts* and cID being a foreign key in relation *Columns*.

(5) for every $l \in L$ and every $f \in F$ with $l \in gran(f)$:
there exists a tuple (l, TRUE, "Dt_<l>", dom(l)) in relation *DimensionLevels* and
a tuple (f, l) in relation *FacthasDim* with f being a foreign key in relation *Facts* and l being a foreign key in relation *DimensionLevels*.

(6) for every $l \in L$ with $\neg\exists\ f \in F$ such that $l \in gran(f)$:
there exists a tuple (l, FALSE, NULL, dom(l)) in relation *DimensionLevels*.

(7) for every $l_1, l_2 \in L$ with $(l_1, l_2) \in$ class:
there exists a tuple (l_1, l_2) in relation *Classifications* with l_1, l_2 being foreign keys in relation *DimensionLevels*.

(8) for every $a \in A$ with attr(a) = l for an $l \in L$:
there exists a tuple (a, l, dom(a)) in relation *Attributes* with l being a foreign key in relation *DimensionLevels*. There exists a tuple (a, cID) in relation *AttributeMapping* with cID being a foreign key in relation *Columns*.

◊

We remark that for a real implementation, certain naming limitations of the DBMS would have to be reflected (see also remark for the example in chapter 4.2). A simple extension of the definition above by a renaming function that replaces illegal characters would solve this issue completely. Since this implementation concept is not necessary for explaining the idea and only increases the complexity of the expressions, we omitted this renaming function here and assumed the same names for both the elements of the conceptual schema, the relational tables and columns, and the contents of the meta schema.

4.4. Transforming Conceptual Schema Evolution Operations to Logical Evolution Operations

In the previous chapters we have introduced the necessary prerequisites for this chapter: the formal mapping between conceptual multidimensional schemas and corresponding relational database schemas on the logical layer. Now, we are able to describe how a sequence of conceptual schema evolution operations can be transformed to corresponding logical evolution operations. As we will see, these logical schema evolution operations adapt the database schema together with the instances and update the contents of the meta schema accordingly.

4.4.1. Overview of Logical Evolution Operations

For processing a given schema evolution job – the semantics of which is specified in terms of the multidimensional data model – on the logical layer, the question arises how do the transformed commands on the logical layer look like? Basically, they are composed of SQL DML and DDL commands. Since we additionally need control structures (loop and branch) that reach beyond the expressiveness of SQL, we describe the logical evolution operations using a description which corresponds to SQL commands embedded into a procedural host language. As already said, their task is to transform the database schema, to adapt the existing instances and to update the meta schema contents. As a consequence, a typical logical evolution operation always consists of three parts:

(1) the transformation of the logical database schema: here, we will introduce abstract relational schema evolution operations that resemble schema evolution (DDL) operations of commercial RDBMS (e.g. `ALTER TABLE ADD COLUMN ...`). Where necessary, the SQL commands will be embedded into a procedural host language (which leads to embedded SQL).

(2) the adaptation of existing instances: this task will be described by SQL DML commands like `UPDATE TABLE Dt_customer SET AGE = ...`. Again, the description will be embedded into a procedural programming language, where necessary.

(3) the update of the meta schema contents: since the structure of the meta schema is fixed, only the contents are updated during a schema evolution. For performing this task, embedded SQL DML commands describe the detailed update semantics (e.g., `INSERT INTO CLASSIFICATIONS ...`) ensuring consistency between schema and contents of the meta schema.

We remark that the logical evolution operations do not necessarily correspond one-to-one to their counterparts on the conceptual layer. Some conceptual schema evolution operations can be directly transformed to a corresponding logical evolution operation whereas in other cases certain sequences of conceptual schema evolution operations are grouped to a complex operation which is then transformed to a logical evolution operation. The reason for this grouping is basically the different expressiveness of star schemas and ME/R models which makes it (in some cases) necessary to collect more information (which is then provided by a group of conceptual schema evolution operations) for the transformation to a corresponding logical evolution operation.

Further, the logical evolution operations are specific for the target system on the logical layer. This means especially they are specific for the logical database schema that is used. The logical evolution operations introduced here are designed for relational star schemas. Corresponding

operations that work with a snowflake schema on the logical layer would look different (i.e. basically, the grouping of the operations and the generated commands on the logical layer).

Before we introduce a set of 14 logical evolution operations[11], we will collect requirements for the transformation algorithm which maps a sequence γ of conceptual schema evolution operations to a sequence λ of logical evolution operations. This sequence λ transforms a consistent logical schema LS to a consistent logical schema LS' (see figure 3-5 in chapter 3.4). In general, the complexity of the transformation arises from the quite different expressiveness of the ME/R model and the star schema. Since the ME/R model is not very restrictive to reflect the modeling requirements of the warehouse designer, a direct mapping of modeling constructs to a star schema is difficult (see also chapters 4.1 and 4.3). The reason for this is the fact that the multidimensional semantics is often hidden or lost in the structure of a star schema. In the following, we will present some cases where the expressiveness strongly differs. Based on these considerations, we will then design an algorithm for the transformation.

4.4.2. Motivating Examples

In order to show some peculiarities of the transformation from which we derive design decisions for the transformation algorithm, we will present some example evolution jobs and discuss their transformation. We assume our standard vehicle repair example with slight variations, i.e. we present partial models that are extended to the complete example or deletions processed on the complete example. We also insert an additional fact *vehicle sales* which allows us the examination of additional interesting cases.

For each motivating example, a figure describes the conceptual evolution job γ and the corresponding logical evolution job λ resulting from the transformation. Another figure shows the modifications of the ME/R model and the resulting modifications of the star schema. We will then discuss the resulting semantics of the operation that adapts the star schema, update the instances and update the meta schema contents (the latter two parts of the operation's semantics are not directly visualized, but contained in the figure which describes the logical evolution job).

Merging Dimensions:

The first example deals with a peculiarity of the ME/R notation that already has been introduced in chapter 4.1.5: a merging dimension.

Since in typical real-world scenarios very often the same classification is used for different dimensions (e.g. customers are classified according to their geographic regions as well as garages, shops, suppliers etc.), the requirement arises to reflect this shared classification in an ME/R model. In figure 4-6, we have seen that garages and customers share the classifications according to geographic region and country. Taking the base levels of the two dimensions (customer and garage) as viewpoint, we could say that the corresponding dimension graphs (represented by the classification hierarchies) are merging. Thus, in such a case, we speak of *merging dimensions* in the ME/R model.

In order to show the processing of new classification edges leading to a merging dimension, we assume that the evolution job consists of an `insert classification` operation between the dimension levels *customer* and *geogr. region*. The conceptual and logical evolution jobs for this case are depicted in figure 4-11. The modifications of the ME/R model and the resulting modifications of the star schema are shown in figure 4-12.

[11] It is a mere coincidence that there are both 14 conceptual schema evolution operations and 14 logical evolution operations. There is no underlying mathematical reason for the number of evolution operations on the two layers.

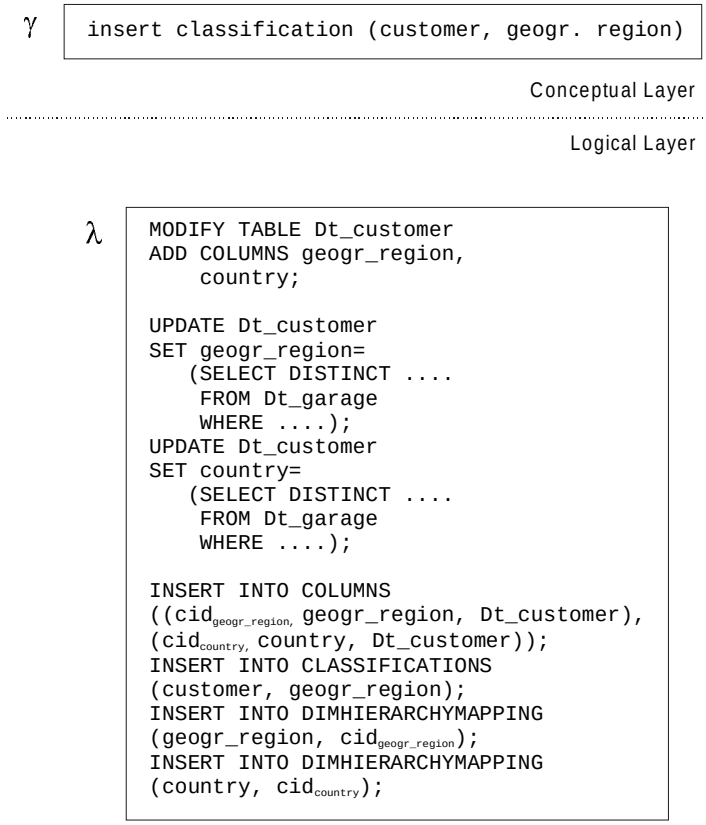


figure 4-11: evolution jobs for a new classification leading to a merging dimension

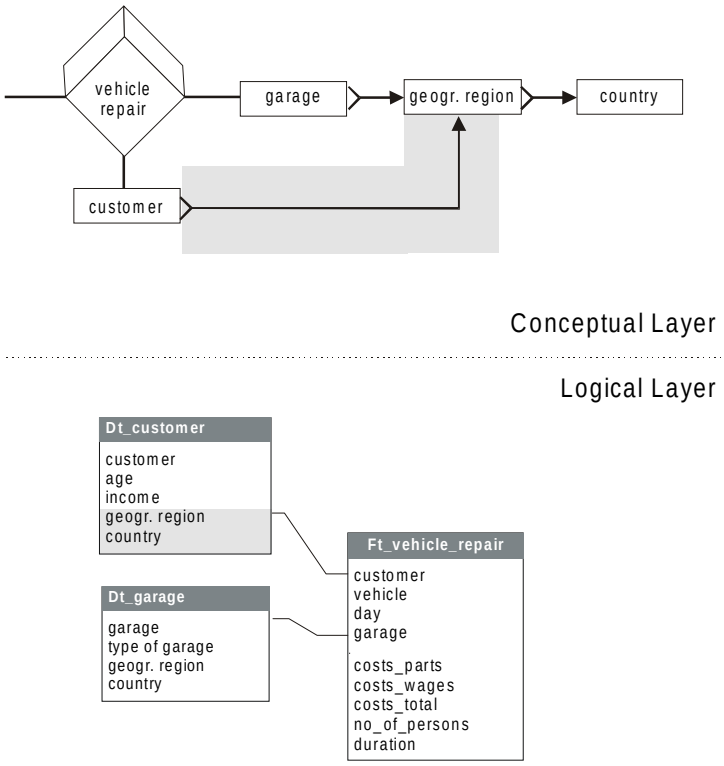


figure 4-12: new classification leading to a merging dimension

How is a merging dimension reflected in a star schema? Since the elements of the corresponding database schema (i.e. tables with columns) cannot share components (which would mean that tables share columns), the corresponding columns have to be duplicated. In the example, the dimension table *Dt_customer* is extended by the columns *geogr. region* and *country* (see figure 4-12 for the resulting database schema, the grey shaded parts mark the modifications).

In general, this means that when a classification edge to an already existing dimension level is defined (in the example from *customer* to *geogr. region*), the transformation algorithm has to check if this dimension level is part of another (already existing) dimension. If so, the corresponding level together with all higher classification levels and describing attributes have to be duplicated in the dimension table to which the source level of the classification edge belongs to.

Analogously to insertions, i.e. duplications of columns, deletions of such merging classification edges lead to removing all duplicated columns in all relevant dimension tables. For example, if we assume that the dimension level *country* would be deleted together with the classification edge from *geogr. region* to *country*, both columns named country in the dimension tables *Dt_customer* and *Dt_garage* would have to be deleted.

The possibly multiple mappings of a dimension level to corresponding columns in different dimension tables are reflected in the meta schema relation *DimHierarchyMapping*.

Conclusion:

A given schema evolution operation may have effects that reach beyond the local elements (i.e., not only the classification edge, but also the dimension levels above this edge in the example) in the ME/R graph. Additionally, there is some kind of redundancy because dimension levels (and also their describing levels) may be mapped to multiple columns in dimension tables

Shared Dimensions:

The next example deals with *shared dimensions*, another peculiarity of ME/R models. Shared dimensions arise when facts share dimensions, possibly at different base levels. We refer to figure 4-14 which shows two alternatives for shared dimensions: either facts share a dimension at the same base level (figure 4-14 a, left side) or at different dimension levels of a classification hierarchy (figure 4-14 b, right side).

In order to present an evolution example, we assume an evolution job which defines a new fact using a shared dimension, more precisely: the evolution job consists of inserting a new fact *vehicle sales* with a new *is_dimension_for* edge between the new fact and an existing level. Let us assume the two different cases of figure 4-14 and let the two facts share the time dimension either at the level *day* (figure 4-14 a) or at the level *month* (figure 4-14 b). The evolution jobs for both cases are depicted in figure 4-13.

In the case of the shared base level *day*, the existing dimension table (*Dt_time*) can be referenced by both fact tables. Concerning schema transformation, this means that a new fact table *Ft_vehicle_sales* is created with a foreign key column referencing day in *Dt_time* (left side of figure 4-13).

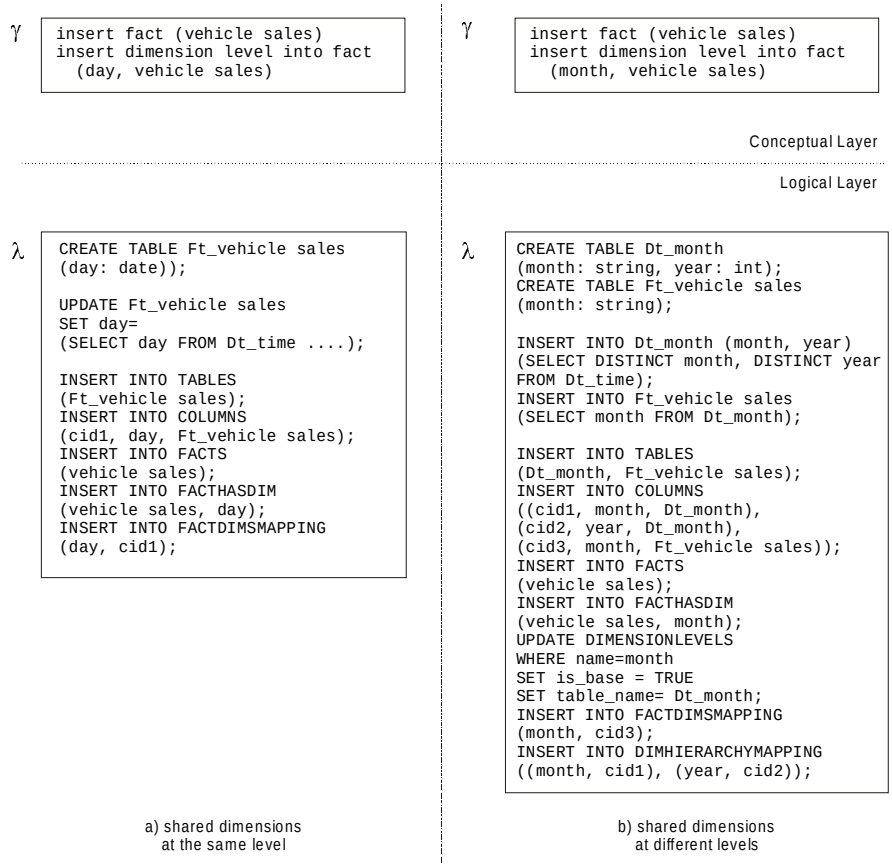


figure 4-13: evolution jobs for a new classification leading to a merging dimension

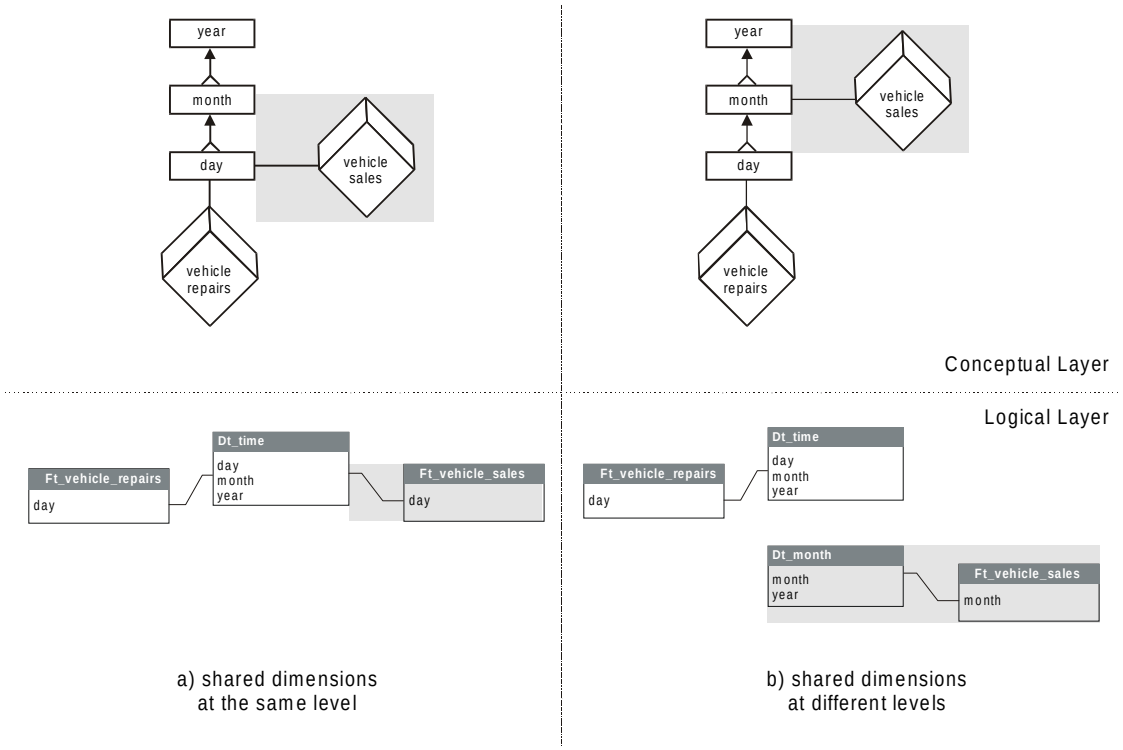


figure 4-14: new fact with shared dimension

In the other case, where vehicle sales shares the time dimension at the level month, a completely new dimension table for the time dimension has to be defined. This new dimension table contains as columns all shared dimension levels above month together with all describing attributes (which does not apply here). Finally, the fact table *Ft_vehicle_sales* references this new dimension table (right side of figure 4-13).

Conclusion:

Although the semantics on the conceptual layer is the same, the resulting semantics of the transformed conceptual operations may be quite different.

Insertions in a classification hierarchy:

As the next example, let us assume the situation that we need to extend an existing dimension hierarchy. To that end, we present a modified version of the vehicle dimension that does not contain the intermediate level vehicle model. We assume an evolution job that inserts this level and thus extends the existing hierarchy by a level. The conceptual and logical evolution job are depicted in figure figure 4-15, the modifications of the ME/R model and the star schema are shown in figure 4-16:

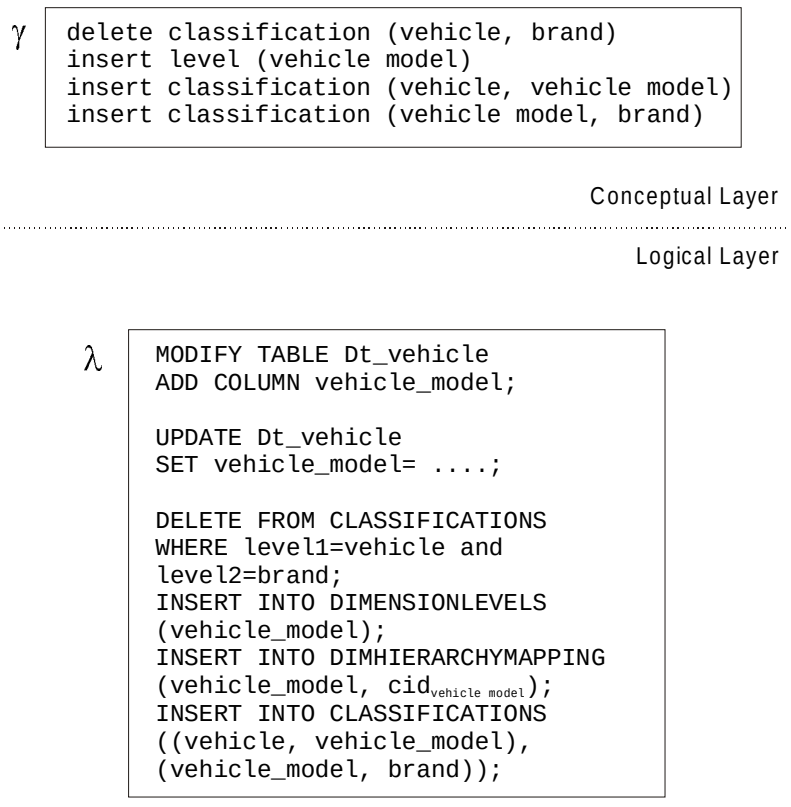


figure 4-15: evolution jobs for insertion in a classification hierarchy

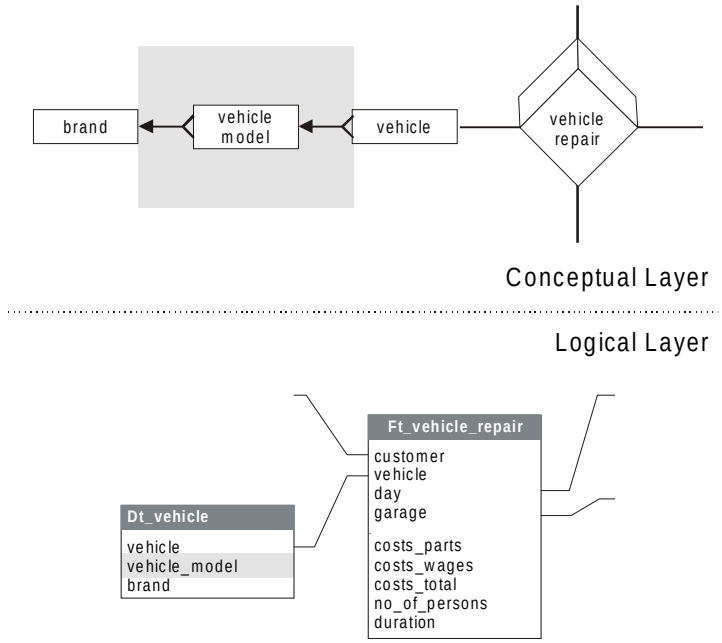


figure 4-16: ME/R model and star schema for insertion in a classification hierarchy

After a `delete classification` operation (the first operation in the example evolution job), it is not yet determined what happens to the left part (i.e. the dimension levels above the second level *brand* in figure 4-16) of the dimension hierarchy. Especially, this partial classification hierarchy may be connected to another dimension or to another fact. The generality of the ME/R approach explicitly allows for such graph transformations which poses complex requirements to the design of the transformation algorithm.

However, as we can see in this example (figure 4-15), it may also happen that the level brand is inserted again later in the same dimension table by the operation `insert classification (vehicle model, brand)`.

In the first case (i.e. the partial classification hierarchy would be connected elsewhere), the transformation algorithm would have to copy the corresponding columns of the dimension table to another table. We have already discussed the copy issue in the example dealing with merging dimensions. In contrast to the situation there (where the specification of the copy issue is hidden in the definition of the *r-up* function), special care has to be taken while copying existing instances due to functional dependencies (and thus redundancies) of the columns (resulting from the unnormalized dimension tables) representing the dimension level hierarchy.

When processing the `delete classification` operation, the transformation algorithm cannot check whether the partial classification hierarchy may be left in the current dimension table or if it is moved to another dimension table. However, in order to perform the instance adaptation, existing instances have to be saved.

As a consequence, we mark the dimension levels above l_2 for deletion when processing a `delete classification` (l_1, l_2) operation. We delete levels physically from a dimension table either when the matching `delete level` operation is processed or at the end of the processing phase. The resulting problem that columns remain in the dimension tables although the corresponding dimension levels may have been moved elsewhere leads to a final garbage collection phase in the transformation algorithm when it is sure that the instances of a level are no longer needed for instance adaptations.

We further remark that the level brand may be contained in more than one dimension (as part of a shared or merging dimension hierarchy). Thus, the algorithm has to check *all* dimension tables in which the level is contained. This motivates the existence of the table *DimHierarchy-Mapping* of the meta schema which represents the information which level is mapped to which column of a dimension table.

Conclusion:

Deletions in a dimension hierarchy may lead to corresponding deletions in the dimension tables. Instances are kept as long as possible in order to avoid loss of information. Since indirectly compensating insertions (re-inserting the same dimension level at another position in the same dimension hierarchy) may be processed later, only a final garbage collection phase (i.e. a traversal for each dimension hierarchy) at the end of the transformation can detect columns to be deleted in the dimension tables.

Alternative Paths in a dimension hierarchy:

As last example, we present a related case (also dealing with classifications) that looks quite easy in the ME/R model, but poses difficult requirements to the transformation algorithm[12], and especially to the garbage collection phase: alternative paths in a dimension hierarchy.

We show the case how to deal with the deletion of a single classification edge in such an alternative path. In figure 4-18, we see the typical alternative path for weeks in the time dimension: days can be classified according to months and weeks, but weeks cannot be classified according to months. Nevertheless, both (calendar) weeks and months can be classified according to years. As evolution job we assume a `delete classification` operation that removes the classification edge between the dimension levels *week* and *year* (figure 4-17). As said before, when processing such an operation (without the corresponding `delete level` *year* operation), all levels above and including year are marked for deletion in dimension table *Dt_time*[13]. Here, we see an exception to this rule: if the level *year* can be reached by another classification path starting at the base level of the dimension, the level *year* must not be deleted from the dimension table. This exception is not easy to handle because the condition cannot be checked locally when processing the `delete classification` operation (neither the base level nor the alternative path are contained in the operation's parameters list).

This interesting exception (which is difficult to detect) refines the specification of the garbage collection phase of the transformation algorithm.

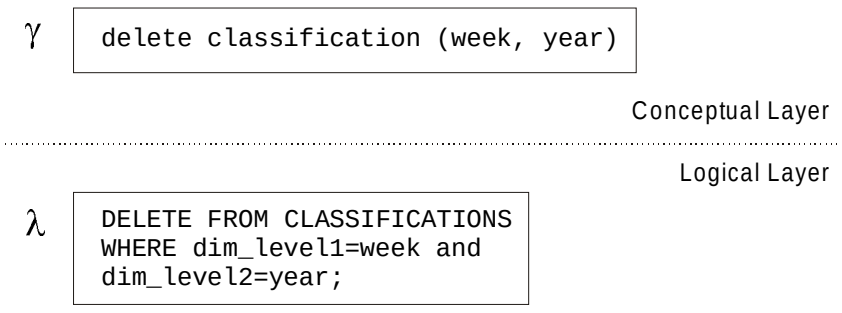


figure 4-17: evolution jobs for deletion of an alternative path

[12] In fact, it even leads to a refinement of the garbage collection specification. See details at the discussion of the transformation algorithm.

[13] In general: year is marked for deletion in all dimension tables, where week belongs to.

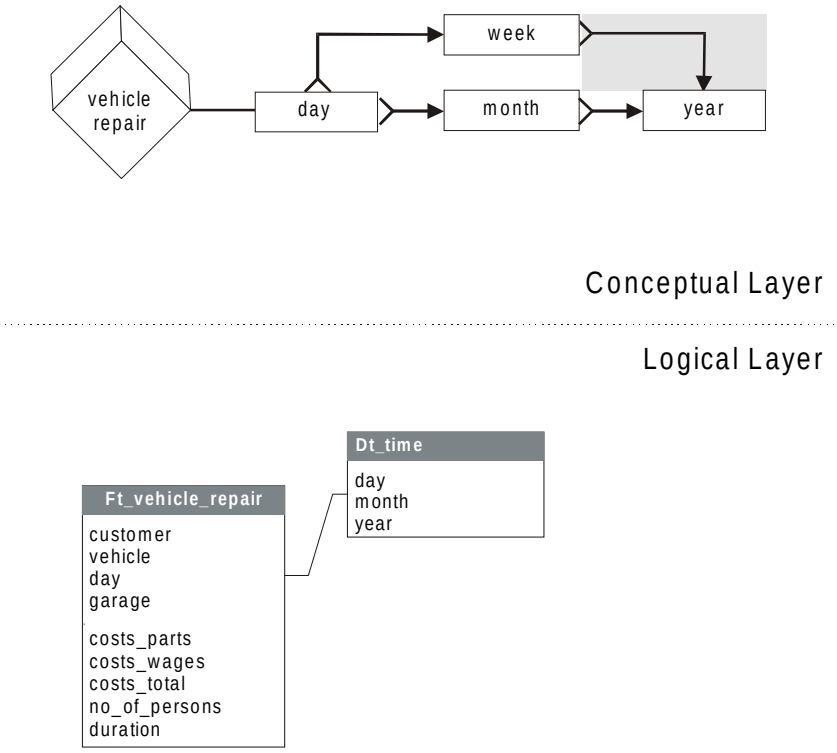


figure 4-18: deleting an alternative path

Conclusion:

The garbage collection has to traverse the full dimension hierarchy graph in order to determine if alternative paths exist.

We will present further examples when explaining the detailed design of the transformation algorithm. As a conclusion, we summarize the lessons learned from these examples: although most schema evolution cases look quite simple in the graphical ME/R model or expressed as evolution job, the processing of such jobs for a star schema is rather complicated. Modifications may have non-local effects that reach beyond a single operation (or its parameters) taking into account additional parts of the ME/R model. Operations with the same semantics on the conceptual layer may be transformed to rather different operations on the logical layer. In order to keep instances for later instance adaptation phases, dimension levels are only marked for deletions. A cleansing step at the end of a processing phase will physically delete marked levels and, as we will see, unreferenced tables.

The next section presents the transformation algorithm. It transforms certain sequences of schema evolution operations to corresponding logical operations. As we will see, sometimes single operations can be transformed directly to a logical evolution operation whereas in other cases more operations (i.e. a part of the sequence) are combined to a complex operation which is then transformed to a corresponding logical evolution operation.

4.4.3. Design of the Transformation Algorithm

The previous chapter collected several requirements for the transformation algorithm. The main issues and identified problem areas arose from the different expressiveness of the ME/R model (or the MD schema, resp.) opposed to the relational star schema. Specifically, the cases of merging dimensions, shared dimensions and alternative hierarchies drove design decisions for the transformation algorithm.

As input for the transformation algorithm, we assume a sequence of conceptual schema evolution operations γ. Formally (see also our formal approach in chapter 3.4), $\gamma = (co_1, co_2, \ldots, co_n)$. As discussed in chapter 3.8, we assume a sequence γ which is minimal in length, transforms a consistent MD schema into another consistent MD schema and is ordered in a way that the pre- and postconditions of the single schema evolution operations hold. Especially, this means concerning deletions that edges are always deleted before the corresponding nodes and concerning insertions that nodes are always inserted before edges connect these nodes.

The output of the transformation algorithm is a sequence of logical evolution operations, denoted as $\lambda = (lo_1, lo_2, \ldots, lo_m)$. This sequence transforms the logical schema, adapts the existing instances and updates the contents of the meta schema accordingly. The basis structure of the transformation algorithm is independent of the target system on the logical layer. The specific parts are the "transformation rules" (which will be explained in the subsequent chapters 4.4.4 and 4.4.5: the single logical evolution operations, their assigned processing priority, their grouping of component conceptual operations, and the generated (SQL) commands).

The base for the design and understanding of the transformation algorithm is the following idea: the algorithm tries to identify applicable sequences of conceptual schema evolution operations which can then be transformed (i.e., re-written) to a corresponding logical evolution operation. Such a sequence consists of one or more conceptual schema evolution operations which we call component operations. A sequence is applicable when its preconditions hold, i.e. more precisely, when certain parameters of the sequence's operations already belong to the MD schema or not (see example below). After transformation, the matching conceptual schema evolution operations are removed from γ. We remark that the component conceptual schema evolution operations of a logical evolution operation need not to be subsequent operations in the sequence γ.

In order to enrich this idea, we present an example for the basic processing model of the transformation algorithm. As starting point, we assume the well-known vehicle repairs example (slightly varied) in figure 4-42 and the evolution job in figure 4-43.

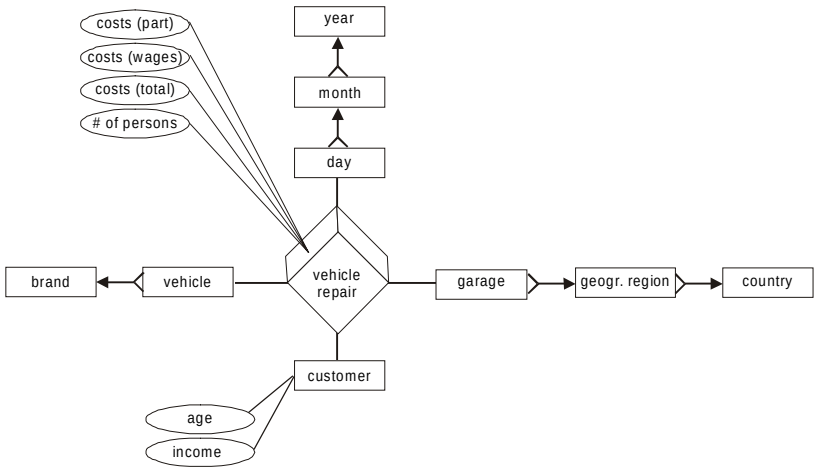


figure 4-19: example schema

γ

```
insert classification (customer,geogr. region);
insert fact (vehicle sales);
insert dimension level into fact
  (day, vehicle sales);
insert attribute (count);
connect attribute to fact (count, vehicle sales);
```

figure 4-20: example evolution job

As said before, the transformation algorithm tries to find applicable operation sequences and transforms these sequences to corresponding logical evolution operations. Refining the observations in chapter 4.4.2, which exemplarily showed the drawbacks of the different expressiveness of the ME/R model and the star schema for the transformation algorithm, we remark that often a single conceptual schema evolution operation does not provide enough information for a transformation to a logical evolution operation. For example, if an attribute is inserted in the ME/R model, but not connected to a fact or a dimension level yet, we do neither know whether to add a column representing the new attribute to a dimension table or a fact table nor to which table the column has to be added (see also our mapping between MD schemas and star schemas in chapter 4.1). As a consequence, we need to transform certain sequences of conceptual schema evolution operations.

In the example evolution job shown in figure 4-20, the sequence `insert fact (vehicle sales); insert dimension level (day, vehicle sales)` provides enough information to be transformed. The corresponding logical evolution command will then basically create a new fact table referring to this dimension table (we refer to chapter 4.4.5 where we refine this example including the logical evolution operations for a star schema). The sequence is also applicable because the existence of the dimension level *day* guarantees that a dimension table containing a column labeled day already exists. We observe that some parameters of the component operations must be matching (i.e., referring to the same fact *vehicle sales*) and others refer to certain elements of the schema (i.e. dimension level *day*). If the dimension level *day* would not be contained in the MD schema yet, the operation sequence would not be applicable because the corresponding dimension table would not exist. As a consequence, we note that the *matching* parameters allow to detect the correct and matching occurrences of the component operations, whereas the *free* (i.e., not matching) parameters provide information about the applicability of the operation sequence. In the example above, we see that the operation sequence `insert attribute (count); connect attribute to fact (count, vehicle sales)` is not applicable in the first step because the fact *vehicle sales* is not contained in the MD schema yet (although the parameter *count* matches in both component operations).

As first step, the transformation algorithm processes the sequence `insert fact (vehicle sales); insert dimension level (day, vehicle sales)`, i.e. it transforms these two conceptual schema evolution operations to a logical evolution operation that transforms the star schema, adapts the instances and updates the meta schema contents (see chapters 4.4.4 and 4.4.5 for details about the generated logical evolution commands). Finally, the two operations are deleted from γ. This processing step is visualized in figure 4-21.

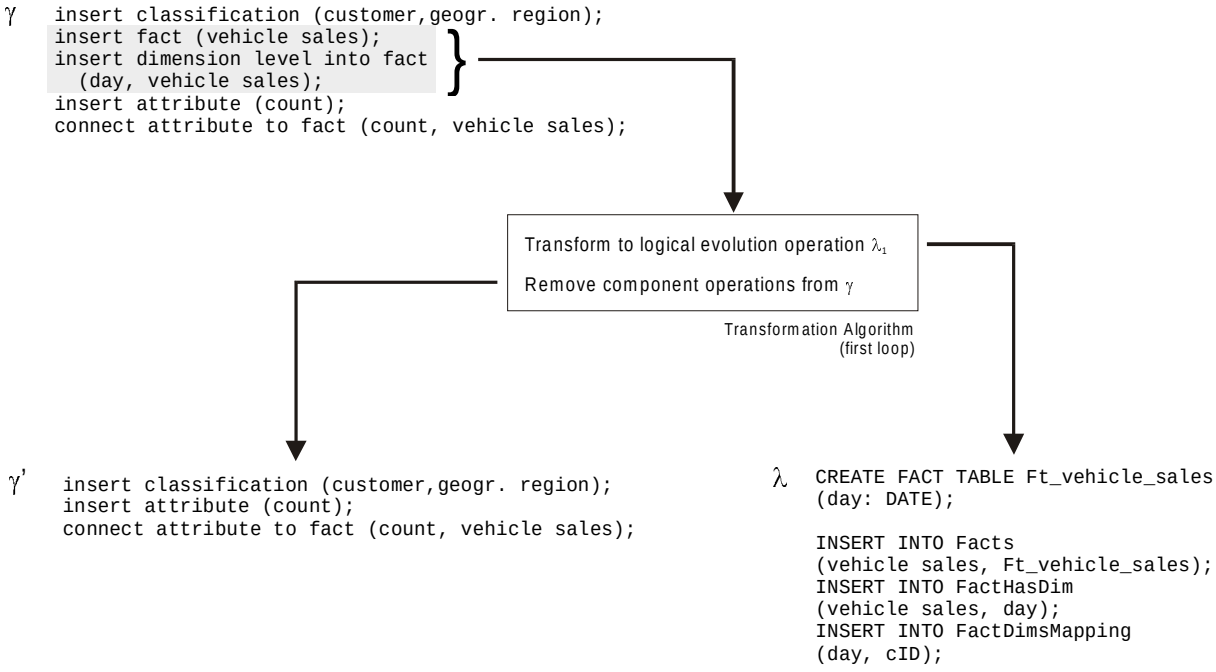


figure 4-21: first transformation loop of the transformation algorithm

Now, as second step, both the sequences `insert classification (customer, geogr. region)` and `insert attribute (count); connect attribute to fact (count, vehicle sales)` are applicable. We will see later that a priority scheme decides about their execution order. After processing the complete evolution job γ, the transformation algorithm holds because γ is empty. The transformation algorithm guarantees a complete and unique parsing of every possible sequence γ. We refer to chapter 4.4.5 where we refine the transformation algorithm (including this example) and discuss these issues in greater detail.

We will now introduce the logical evolution operations. The basic idea is to comprehend as many conceptual schema evolution operations as necessary (i.e. a sequence that provides enough semantics and context for a transformation) and to consider them as a complex operation which then can be transformed to a logical evolution operation. Chapter 4.4.5 will then present the overall transformation algorithm together with the different priority classes of the sequences and their corresponding logical evolution operations. We will also provide more details about the garbage collection step there.

4.4.4. Logical Evolution Operations

As introduced in chapter 3.8, we assume that during a schema maintenance session in the graphical schema design tool [SBH00] a sequence of conceptual schema evolution operations is applied to a given MD schema which is visualized by its ME/R graph (see chapter 3.6). After having given the base idea of the transformation algorithm, we now introduce the corresponding logical evolution operations to which a sequence of conceptual schema evolution operations is transformed.

We remark that the logical evolution operations are designed for star schemas on the logical layer. When using a different schema template on the logical layer (e.g. snowflake schema), the

logical evolution operations would be different. See chapter 5 for an extended discussion of this issue.

First, an overview of the fourteen logical evolution operations is given (figure 4-22). For each logical evolution operation, the corresponding composition of conceptual schema evolution operations is shown as ME/R diagram. In order to distinguish logical from conceptual operations, logical operations always have an 'L' as subscript (e.g.*insert measure column*$_L$) The modified parts are depicted in grey.

The first question that arises when regarding figure 4-22 is: why exactly these operations? The general answer is that we have selected these operations because they adequately reflect the different expressiveness of an MD schema and its corresponding relational star schema. More precisely, the combinations of conceptual schema evolution operations are chosen in a way that they can be transformed to a corresponding, semantically meaningful, modification of the star schema. We explain this in more detail by giving the argumentation that led to each operation.

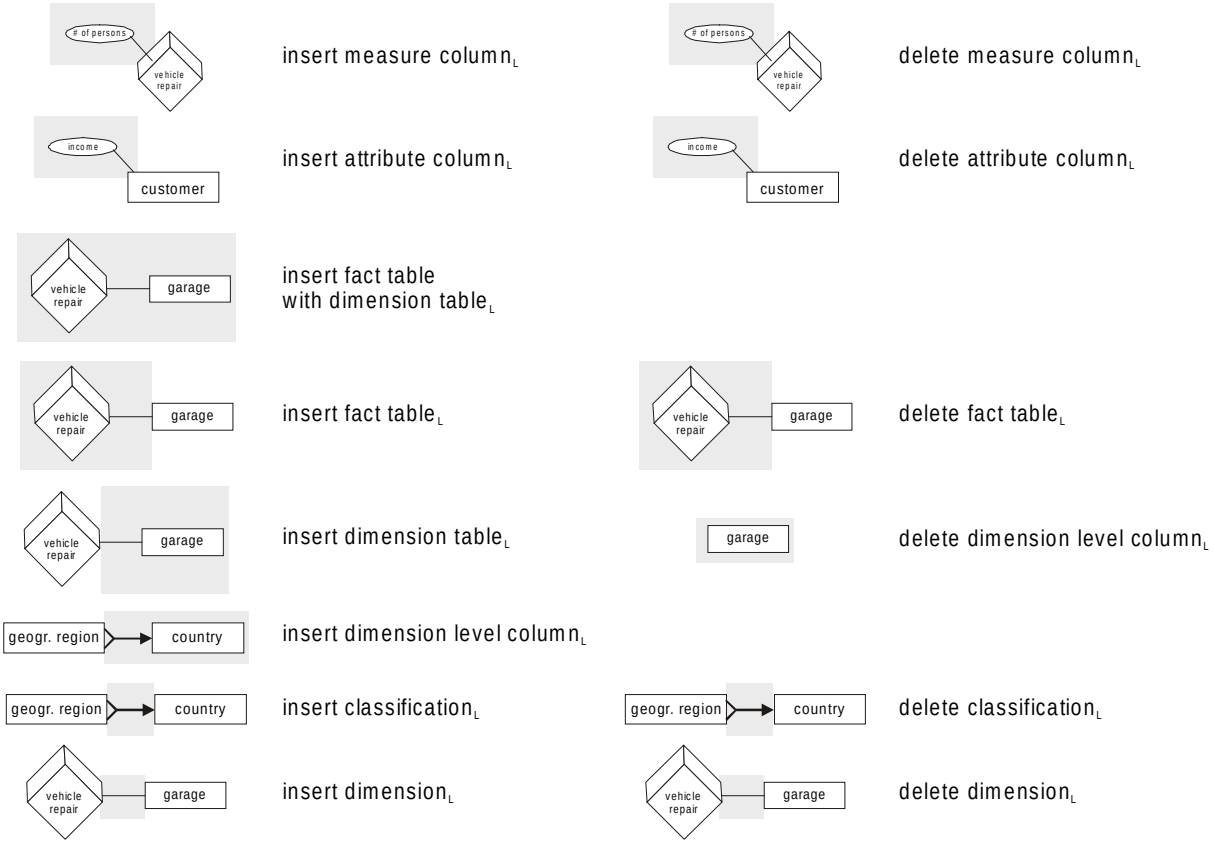


figure 4-22: overview of the logical evolution operations

First, the design of the four operations dealing with attributes is rather straightforward. An isolated attribute does not provide enough semantics, it can merely mapped to a column in a relational table. Since it is still unspecified whether it is mapped to a column of a dimension table or a fact table, it must be stated (which is then done in terms of another operation) where it belongs to, or graphically, where it is connected to. This information allows the precise map-

ping to either a dimension table or a fact table[14]. Since attributes can only be connected to facts as measure attributes (and then become columns of fact tables) or to dimension levels as describing attributes (and then become columns of the corresponding dimension tables), we need this additional *connect attribute to* operation that binds the attribute to either a fact or a dimension level. This explains the ideas that led to the design of the four operations `insert measure column`$_L$, `delete measure column`$_L$, `insert attribute column`$_L$, `delete attribute column`$_L$.

The next operation that contains enough semantics to be adequately mapped to a star schema is the operation `insert fact table with dimension table`$_L$. On the conceptual layer, it consists of the insertion of a fact, a dimension level, and a corresponding *is_dimension_of* edge between the fact and the dimension level. When considering the star schema, it is clear that only the composition of these three operations provides enough semantics and context to build a corresponding star schema consisting of a new dimension table and a corresponding new fact table.

The case is different if

- either the dimension level already exists (which corresponds to the existence of the dimension table) and the fact together with the *is_dimension_of edge* is inserted
- or the fact already exists (which corresponds to the existence of the fact table) and the dimension level together with the *is_dimension_of edge* is inserted.

The first case leads to the operation `insert fact table`$_L$, the second to the operation `insert dimension table`$_L$. In both cases, there is already one part reflected in a relational table (either a dimension table or a fact table) and the other part is newly inserted which then leads to the creation of the counterpart table in the star schema. We remark that albeit in the specification of the operation `insert fact table`$_L$ there are two quite different cases to be considered, we decided to provide only a single operation because both cases have the same composition of conceptual schema evolution operations.

Concerning deletions, the operations for these cases discussed above look rather different. There is no counterpart to the operation `insert fact table with dimension table`$_L$ because deletions can be performed with a smaller granularity. The reason is that when deleting elements, the necessary semantics is provided by a shorter context (under context we understand a combination of conceptual schema evolution operations with their parameters) which is reflected in shorter combinations of conceptual schema evolution operations. Therefore, we only have an operation `delete fact table`$_L$ that removes the smallest possible fact table (fact table with a single foreign key attribute referencing the dimension table) corresponding to the deletion of a fact together with the corresponding *is_dimension_of* edge. Similarly, we only need an operation `delete dimension level column`$_L$ that deletes the column representing a dimension level in all matching dimension tables which corresponds to the deletion of an isolated (i.e., not connected via any edges) dimension level. The corresponding operations removing edges to and from dimension levels will be explained below.

Since a dimension level may be connected to many different facts and dimension levels by corresponding edges (see for example the level *geogr. region* in figure 4-23), these fine-grained operations offer the additional advantage that the number of logical evolution operations can be reduced: an operation that deletes an isolated dimension level together with operations deleting all edges that may be connected to an dimension level is enough. Otherwise, we would

[14] We remark that the general problem arises from the fact that columns are second class citizens of the relational model (i.e., they cannot exist alone, but must always be defined as part of tables), whereas attributes are first-class citizens of the ME/R model.

additionally have to provide all possible groupings as operations (deletions of dimension level together with all possible edge types) which would increase the number of operations.

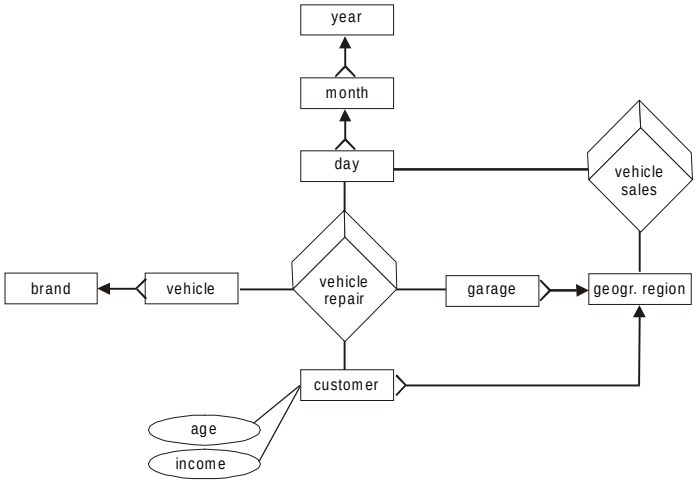


figure 4-23: the advantage of fine-grained deletion operations

Concerning insertions of classification edges, there are two possible cases and therefore we also offer two corresponding operations. The first is the insertion of a classification edge together with the target level (`insert dimension level column`$_L$) which corresponds to the definition of a new highest level in a classification hierarchy, a case which often arises in practical schema maintenance. The second case leads us to an interesting observation:

So far, the names of the logical evolution operations have been closely related to the corresponding modification of the star schema. This was due to the fact that the combination of conceptual schema evolution operations provided enough information (expressed by the operations with their parameters) to transform this sequence to a corresponding consistent star schema modification. The operation could then be named according to this modification.

The last four logical evolution operations that we introduce consist only of a single conceptual schema evolution operation. We need them to express our special cases of merging and shared dimensions. Here, the operation alone does not deliver the necessary context for a direct transformation into a star schema modification (plus instance adaptation and meta schema contents update). Therefore, for the description of the semantics, different cases have to be reflected which then lead to different star schema modifications. Consequently, we could not choose names for these operations that remind of the semantics of the star schema transformation (because there were rather different types of modifications). Therefore, we selected names that resemble the names of the conceptual schema evolution operation, as we will see now.

The second case for classification edges is the insertion of only the classification edge alone (`insert classification`$_L$). This operation is needed when both dimension levels already exist. This may be the case when either merging dimensions are defined, or in case a dimension level is disconnected (either from a fact or from a classification hierarchy) during an evolution job and then connected to another dimension level leading to a new hierarchy. Regarding deletions of classification edges, we again only need the case that a dimension level is deleted (as already explained above) and the case that a classification edge alone is deleted (delete `classification`$_L$).

Finally, two operations that insert or delete, resp., an *is_dimension_of* edge (when both the fact and the dimension level already exist or still exist, resp.) are needed (`insert dimension`$_L$, `delete dimension`$_L$). The reasons for these two operations are shared dimensions of a fact or

again the situation that a dimension level has been disconnected during an evolution job and is then connected to a fact.

After having motivated the single logical evolution operations, we now describe their semantics.

For each logical evolution operation, the following information is provided:

- the name of the operation
- its composition of component conceptual schema evolution operations (with references to the corresponding operations in chapter 3.7). This corresponds to the sequence of operations which is searched in γ for identifying applicable operations.
- its parameters (see also chapter 3.7 for corresponding parameters of the conceptual schema evolution operations)
- preconditions for the execution of the operation. These preconditions are specified in terms of the logical (database) layer, thus we call them logical preconditions. The logical preconditions must be fulfilled in order to ensure correct execution of the logical evolution operations. Due to the design of our mapping between the conceptual and logical layer (see chapters 4.1 and 4.3), the logical preconditions can be checked using the preconditions of the component conceptual schema evolution operations. The validity of the conceptual preconditions is ensured by the graphical modeling environment and, consequently, the preconditions for the logical operations are fulfilled by the correct ordering of the conceptual operation sequence. We remark that the preconditions depend on the target system on the logical layer.
- an informal description of the operation's semantics including schema transformation, instance adaptation, and update of meta schema contents (listed under three dots).

For a complete formal specification of the semantics of each operation, we refer the reader to Appendix B. In order to distinguish the logical evolution operations from the conceptual schema evolution operations, we denote logical evolution operations with an "L" (for "logical") as subscript, e.g. *insert measure column*$_L$.

1. **insert measure column**$_L$: this operation extends an existing fact table by a new measure column.

insert measure column$_L$	
Composition: (see figure 3-12;figure 3-16)	insert attribute (m); connect attribute to fact (m,f, g)
Parameters:	measure m, fact f, instance adaptation function g
Preconditions:	The fact table Ft_<f> exists and does not contain a column labeled m (assured by the conceptual preconditions: $f \in F$, $m \notin A$)
Semantics:	▪ The fact table Ft_<f> is extended by the new column m. ▪ The values of measure m are updated using the instance adaptation function g.

	▪ A new tuple representing m is inserted in the meta schema relation *Measures* and a new tuple representing the new column m in the fact table is inserted in relation *Columns*[15].

figure 4-24: operation *insert measure column*$_L$

2. **delete measure column$_L$**: this operation deletes a measure column from a fact table.

delete measure column$_L$	
Composition: (see figure 3-17;figure 3-13)	disconnect attribute from fact (m,f); delete attribute (m)
Parameters:	attribute m, fact f
Preconditions:	The fact table Ft_<f> exists and contains a column labeled m (assured by the conceptual preconditions: $f \in F$, $m \in A$, attr(m) = f)
Semantics:	▪ The column m is removed from the fact table Ft_<f>. ▪ Instance adaptation is done implicitly by schema transformation (deletion of the instances) ▪ The tuple representing m is deleted from relation *Measures* and the tuple representing the deleted column m in the fact table is deleted from relation *Columns*.

figure 4-25: operation *delete measure column$_L$*

3. **insert attribute column$_L$**: this operation extends an existing dimension table by a new describing attribute column of a dimension level.

insert attribute column$_L$	
Composition: (see figure 3-12;figure 3-14)	insert attribute (a); connect attribute to dim_level (a,l,g)
Parameters:	attribute a, dimension level l, instance adaptation function g

[15] We remark that updates to the system catalogue relations *Columns* and *Tables* are done by the DBMS when processing the corresponding DDL command to transform the logical database schema.

Preconditions:	At least one dimension table Dt_<bl> exists which contains a column for dimension level l. Each Dt_<bl> does not contain yet a column labeled a (assured by the conceptual preconditions: $l \in L$, $a \notin A$)
Semantics:	▪ Since the dimension level l may be a shared level of more dimensions and thus contained in more then one dimension table: add a new column a to all dimension tables Dt_<bl> to which l belongs (this step includes finding the relevant base level bl for dimension level l). ▪ for all dimension tables to which l belongs: update the values for the new column a using the instance adaptation function g. ▪ for all dimension tables to which l belongs: insert a new tuple representing the new column in the dimension table in relation *Columns*. Insert a new tuple representing a in relation *Attributes*. Insert new tuples representing the mapping of attribute a to the corresponding columns in all matching Dt_<bl> in the relation *Attributemapping*.

figure 4-26: operation *insert attribute column*$_L$

4. **delete attribute column**$_L$: deletes a describing attribute column from a dimension table.

delete attribute column$_L$	
Composition: (see figure 3-13;figure 3-15)	disconnect attribute from dim_level (a,l); delete attribute (a)
Parameters:	attribute a, dimension level l
Preconditions:	At least one dimension table Dt_<bl> exists which contains a column for dimension level l and a column for attribute a (assured by the conceptual preconditions: $l \in L$, $a \in A$, attr(a) = l)
Semantics:	▪ delete column a from all dimension tables (denoted by Dt_<bl>) to which l belongs. ▪ instance adaptation is done implicitly by schema transformation (deletion of the instances). ▪ delete all tuples referencing column a in meta schema relation *Columns*. Delete all tuples referencing attribute a in relation *Attributes*. Delete all tuples representing the mapping from attribute a to its corresponding column in a dimension table in relation *AttributeMapping*.

figure 4-27: operation *delete attribute column*$_L$

5. **insert fact table with dimension table$_L$**: this operation inserts a new fact table together with a new dimension table which is referenced by the fact table (i.e. the dimension level is base level of the fact).

insert fact table with dimension table$_L$	
Composition: (see figure 3-10; figure 3-20; figure 3-23)	insert fact (f); insert level (l); insert dimension into fact (f, l)
Parameters:	fact f, dimension level l
Preconditions:	Neither a dimension table named Dt_<l> nor a fact table named Ft_<f> exist (assured by the conceptual preconditions: $l \notin L$, $f \notin F$)
Semantics:	▪ create a new dimension table Dt_<l> with column l and a new fact table Ft_<f> with column l referencing Dt_<l>. ▪ Since both the dimension level and the fact are new, there are no *existing* instances to be adapted. We assume that new instances are inserted outside the scope of our schema design task. ▪ Insert two tuples for the the two new tables in meta schema relation *Tables*. Insert two tuples for the new columns of these new tables in relation *Columns*. Insert a tuple for the new dimension level in relation *DimensionLevels* and a tuple for the new fact in relation *Facts*. Insert a tuple in relation *FactHasDim* indicating that l forms the base level for a new dimension of fact f. Insert a new tuple in relation *FactDimsMapping* indicating that level l is mapped to the corresponding column in the fact table Ft_<f>. Insert a new tuple in relation *DimHierarchyMapping* indicating that level l is mapped to the corresponding column in Dt_<l>

figure 4-28: operation *insert fact table with dimension table$_L$*

6. **insert fact table$_L$**: this operation inserts a new fact table and relates it to an existing dimension table.

insert fact table$_L$	
Composition: (see figure 3-20;figure 3-23)	insert fact (f); insert dimension into fact (f, l, nv)
Parameters:	fact f, dimension level l, instance adaptation function nv

Preconditions:	A dimension table exists that contains a column representing dimension level l. l may be the base level (i.e., the dimension table is labeled Dt_<l>) or not (i.e., the dimension table is labeled Dt_<bl>). No fact table labeled Ft_<f> exists. This is assured by the conceptual preconditions $l \in L$, $f \notin F$.
Semantics:	▪ Case 1: l is (or has been) base level of another fact: create a new fact table Dt_<f > with column l referencing the appropriate dimension table. Case 2: l is not base level of any fact: first, create a new dimension table Dt_<l> with all levels and their describing attributes above (and including) l in the classification hierarchy (i.e. copy all levels and their attributes to this new dimension table). Second, create a new fact table Dt_<f > with column l referencing the new dimension table Dt_<l>. ▪ Case 1: l is (or has been) base level of another fact: Since the fact table is new, there are no *existing* instances to be adapted. Case 2: a new dimension table Dt_<l> has been created: copy all distinct values for l into column l of Dt_<l>. Copy the values for all higher levels including the values for all describing attributes of these levels (incl. l) in the corresponding columns of Dt_<l>. ▪ Case 1: l is (or has been) base level of another fact: insert a new tuple into relation *Tables* for the new fact table and a new tuple for the new column of this fact table in relation *Columns*. Insert a new tuple for the new fact in relation *Facts* and a corresponding new tuple in relation *FactHasDim* describing that l is a base level spanning a dimension for fact f. Insert a new tuple in relation *FactDimsMapping* indicating that level l is mapped to the corresponding column in the fact table Ft_<f>. Insert a new tuple in relation *DimHierarchyMapping* indicating that level l is mapped to the corresponding column in Dt_<l> Case 2: l is not base level of any fact: **additionally** to case 1, perform the following steps: insert a new tuple into relation *Tables* for the newly created dimension table Dt_<l>. Insert new tuples in relation *Columns* for all dimension levels and their attributes which have been inserted as columns in this new dimension table. Update relation *DimensionLevels*: set the *is_base* flag for level l to TRUE and set Dt_<l> as table_name.

figure 4-29: operation *insert fact table*$_L$

7. **delete fact table$_L$**: this operation deletes an existing fact table. The dimension table to which the fact table was related still exists after execution of this operation.

delete fact table$_L$	
Composition: (see figure 3-24;figure 3-21)	delete dimension (l,f,agg); delete fact (f)
Parameters:	fact f, dimension level l. The aggregation function agg is not used because the fact is deleted, too.
Preconditions:	A fact table labeled Ft_<f> with a single column l referencing the corresponding dimension table Dt_<l> exists (assured by the conceptual preconditions $l \in L$, $f \in F$, $l \in gran(f)$). The fact must not be connected to any other elements than the dimension level l. This precondition is guaranteed because a delete fact operation may only occur after the last edge is deleted from this fact (see remark concerning the ordering of schema evolution operations)
Semantics:	▪ Drop the fact table Ft_<f>. ▪ Instance adaptation (i.e. deletion) is done implicitly by the schema transformation. ▪ Delete all tuples referencing Ft_<f> from relations *Tables* and *Columns*. Delete the tuple referencing fact f in relation *Facts*. Delete the tuple referencing fact f and dimension level l from relation *FactHasDim*. Delete the appropriate tuple from relation *FactDimsMapping* indicating that dimension level l was mapped to the corresponding column in fact table Ft_<f>.

figure 4-30: operation *delete fact table$_L$*

8. **insert dimension table$_L$**: this operation inserts a new dimension table and relates it with an existing fact table.

insert dimension table$_L$	
Composition: (see figure 3-10;figure 3-23)	insert level (l); insert dimension into fact (f, l)
Parameters:	fact f, dimension level l
Preconditions:	A fact table Ft_<f> exists and does not yet contain a column named l (assured by the conceptual preconditions: $l \notin L$, $f \in F$)
Semantics:	▪ Create a new dimension table Dt_<l> with column l. Add

	a column l (referencing Dt_<l>) to the fact table Ft_<f>. ■ Since the dimension level is new, there are no *existing* instances in the fact table or dimension table to be adapted. We assume that new instances are inserted outside the scope of the schema design task. ■ Insert a new tuple into relation *Tables* describing the new dimension table. Insert two new tuples in relation *Columnns* describing both the new column in the new dimension table and the new column in the existing fact table. Insert a new tuple in relation *DimensionLevels* for the newly inserted dimension level. Insert a new tuple in relation *FactHasDim* indicating that l is a base dimension level for fact f. Insert a new tuple into relation *FactDimsMapping* indicating that l is a foreign key attribute in the fact table Ft_<f>. Insert a new tuple in relation *DimHierarchyMapping* indicating that l is mapped to the corresponding column in Dt_<l> .

figure 4-31: operation *insert dimension table*$_L$

9. **insert dimension level column**$_L$: this operation inserts a new column representing a new dimension level into an existing dimension table. As a consequence of this operation's execution, the new level is (maybe temporarily) the highest level of the classification hierarchy (i.e. after insertion of the level, there exists a path from the base level of the dimension to the new level. The path ends at the new level).

insert dimension level column$_L$	
Composition: (see figure 3-10;figure 3-18)	insert level (l_2); insert classification (l_1, l_2, $r-up_{l1}^{l2}$)
Parameters:	dimension levels l_1, l_2, instance adaptation function $r-up_{l1}^{l2}$. Level l_1 is the level to which the new level l_2 is being connected.
Preconditions:	At least one dimension table Dt_<bl> exists which contains a column representing l_1, but no column named l_2 (assured by : $l_1 \in L$, $l_2 \notin L$)
Semantics:	■ Add a column l_2 to all dimension tables (denoted by Dt_<bl>) in which l_1 is contained. ■ For all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: update the new column l_2 using the roll-up function $r-up_{l1}^{l2}$. ■ Insert a new tuple in relation *DimensionLevels* representing the newly inserted dimension level l_2. Insert a new tuple (l_1,l_2) in relation *Classifications* representing the

	new classification relationship. Additionally: for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: insert a new tuple in relation *Columns* which represents the new column l_2 of Dt_<bl>. Insert a new tuple into relation *DimHierarchy-Mapping* which describes the mapping from l_2 to each of these columns in the dimension table(s) Dt_<bl>.

figure 4-32: operation *insert dimension level column*$_L$

10. insert classification$_L$: this operation inserts a new classification relationship between two existing dimension levels.

insert classification$_L$	
Composition: (see figure 3-18)	insert classification (l_1, l_2)
Parameters:	dimension levels l_1, l_2 to be connected. The new classification relationship means that level l_1 can be classified according to level l_2.
Preconditions:	There exists at least one dimension table which contains a column named l_1 . There exists also at least one dimension table (possibly the same) which contains a column named l_2 . This is assured by the conceptual preconditions: $l_1 \in L$, $l_2 \in L$.
Semantics:	▪ for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: Case 1: l_2 exists already in Dt_<bl>: if l_2 is marked for deletion: unset deletion flag Case 2: l_2 exists in another dimension table: Add l_2 and all dimension levels above l_2 in the classification hierarchy together with all describing attributes of these levels as new columns to Dt_<bl>. ▪ for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: Case 1: l_2 exists already in Dt_<bl>: no instance adaptation necessary. Case 2: l_2 exists in another dimension table: update all copied levels (i.e. l_2 and all levels above in the classification hierarchy) using the corresponding *r-up* functions. Update all de-

	scribing attributes of these dimension levels using the corresponding av_a function. ▪ for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: insert the tuple (l_1,l_2) in relation *Classifications*. Case 1: l_2 exists already in Dt_<bl>: done. Case 2: l_2 exists in another dimension table: for each of the copied dimension levels: insert a corresponding tuple in relation *Columns* and a corresponding tuple in relation *DimHierarchy-Mapping*. for each of the copied attributes of these levels: insert a corresponding tuple in relation *Columns* and a corresponding tuple in relation *Attribute-Mapping*.

figure 4-33: operation *insert classification*$_L$

11. **delete classification**$_L$: this operation removes an existing classification relationship between two dimension levels. Only the classification edge is deleted, the dimension levels still exist after execution of this operation.

delete classification$_L$	
Composition: (see figure 3-19)	delete classification (l_1, l_2)
Parameters:	dimension levels l_1, l_2 to be disconnected.
Preconditions:	There exists at least one dimension table which contains a column labeled l_1 and at least one dimension table which contains a column labeled l_2 (assured by the conceptual preconditions $l_1 \in L$, $l_2 \in L$, (l_1, l_2) $\in$ class).
Semantics:	▪ for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: mark all dimension levels above and including l_2 (i.e. all levels $\in D_{l2}$) and all attributes of these levels (i.e. all attributes $\in$ Attributes(D_{l2})) for deletion. ▪ No instance adaptation when processing this operation. The instances of l_2 are either needed again when this level is connected elsewhere, deleted when l_2 is deleted, or deleted from the dimension table(s) Dt_<bl> during the garbage collection at the end of the processing phase in the

	transformation algorithm. ■ Delete the tuple (l_1,l_2) in relation *Classifications*. Additionally, for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: mark all tuples representing dimension levels above and including l_2 and all attributes of these levels for deletion in relation *Columns*. Mark all tuples representing dimension levels above and including l_2 in relation *DimHierarchyMapping* for deletion. Mark all tuples representing attributes of these dimension levels in relation *AttributeMapping* for deletion.

figure 4-34: operation *delete classification*$_L$

12. delete dimension level column$_L$: deletes a column representing a dimension level in an existing dimension table.

delete dimension level column$_L$	
Composition: (see figure 3-11)	delete level (l)
Parameters:	dimension level l
Preconditions:	At least one dimension table exists which contains a column labeled l. l may be the base level of this dimension table (i.e. the dimension table is named Dt_<l>). This is assured by the conceptual precondition l ∈ L.

Semantics:	■ for all dimension tables (denoted by Dt_<bl>) to which l belongs:
	If (l is the only level of this dimension table)
	then delete the dimension table Dt_<bl>
	else delete the column l in Dt_<bl>
	■ for all dimension tables (denoted by Dt_<bl>) to which l belongs:
	If (l is the base level of this dimension table) and
	(l is not the only level of Dt_<bl>)
	then eliminate duplicates in the new base level
	(All other cases are done implicitly by the schema transformation).
	■ for all dimension tables (denoted by Dt_<bl>) to which l belongs:
	If (l is the only level of this dimension table)
	then

	delete the tuple representing l in Dt_<l> in *Columns*; delete the tuple representing Dt_<l> in *Tables*; delete the tuples representing l in *FactDimsMapping* else if (l is base level, but not the only level of this dimension table) then delete the tuple representing l in Dt_<l> in *Columns*; delete the tuples representing l in *FactDimsMapping*; else delete the tuple representing l in Dt_<l> in *Columns*; Additionally in all cases: delete the tuples representing l in *DimHierarchyMapping*; delete the tuple representing l in *DimensionLevels*;

figure 4-35: operation *delete dimension level column*$_L$

13. insert dimension$_L$: this operation inserts a new *is_dimension_of* edge between an existing fact and an existing dimension level which is then base level of the fact.

insert dimension$_L$	
Composition: (see figure 3-23)	insert dimension into fact (f, l, nv)
Parameters:	fact f, dimension level l, instance adaptation function nv
Preconditions:	There exists at least one dimension table containing a column labeled l. The dimension table may be labeled Dt_<l> (if l is the base level) or Dt_<bl> (if l is not the base level). There exists a fact table Ft_<f> which does not yet contain a column labeled l (assured by the conceptual preconditions: $l \in L$, $f \in F$)
Semantics:	▪ Case 1: l is (or has been) base level of another fact: add a column l referencing the appropriate dimension table to the fact table Ft_<f >. Case 2: l is not base level of any fact: first, create a new dimension table Dt_<l> with all levels and their describing attributes above (and including) l in the classification hierarchy (i.e. copy all levels and their attributes to this new dimension table). Second, add a column l referencing the new dimension table Dt_<l> to the fact table Ft_<f >. ▪ Start with case 2: if a new dimension table Dt_<l> has been created: copy all distinct values of l into column l of

	Dt_<l>. Copy the values for all higher levels including the values for all describing attributes of these levels (incl. l) in the corresponding columns of Dt_<l>. In both cases: adapt the data in the fact table Ft_<f> according to the increased dimensionality. To that end, define a temporary table with attributes $FK_f \cup l \cup Measure_f$. Insert into the temporary table the cross-product of Ft_<f> and the attribute l from Dt_<l>. Update all measure attributes using the instance adaptation function nv. Replace data in the fact table Ft_<f> by all not-NULL values from the temp table. ▪ Case 1: l is (or has been) base level of another fact: insert a new tuple for the new column of the fact table in relation *Columns*. Insert a corresponding new tuple in relation *FactHasDim* describing that l is a base level spanning a dimension for fact f. Insert a new tuple in relation *FactDimsMapping* indicating that level l is mapped to the corresponding column in the fact table Ft_<f>. Insert a new tuple in relation *DimHierarchyMapping* indicating that level l is mapped to the corresponding column in Dt_<l> Case 2: l is not base level of any fact: **additionally** to case 1, perform the following steps: insert a new tuple into relation *Tables* for the newly created dimension table Dt_<l>. Insert new tuples in relation *Columns* for all dimension levels and their attributes which have been inserted as columns in this new dimension table. Update relation *DimensionLevels*: set the *is_base* flag for level l to TRUE and set Dt_<l> as table_name.

figure 4-36: operation *insert dimension*$_L$

14. delete dimension$_L$: this operation deletes an existing *is_dimension_of* edge between a fact and a dimension level. Both the fact and the dimension level still exist after execution of the operation.

delete dimension$_L$	
Composition: (see figure 3-24)	delete dimension level from fact (l, f, agg)
Parameters:	fact f, dimension level l, instance adaptation function agg

Preconditions:	There exists a dimension table named Dt_<l> containing a column named l. There exists a fact table Ft_<f> with a column l referencing Dt_<l>. This is assured by the conceptual preconditions: $l \in L$, $f \in F$, $l \in gran(f)$.
Semantics:	▪ We remark that the instance adpatation must be executed before the schema transformation takes place. Otherwise, necessary instance information would be lost. The column l is removed from the fact table Ft_<f>. ▪ The data in the fact table has to be adapted according to the decreased dimensionality, i.e. the measure values have to be aggregated using the instance adaptation function agg. To that end, a copy of the fact table to a temporary table is done. The dimension level to be deleted is excluded and the measure attributes are aggregated using the instance adaptation function. After aggregation, the data in the fact table is replaced by the contents of the temporary table. ▪ The tuple representing the mapping of l to the corresponding column in Ft_<f> must be deleted from the relation *FactDimsMapping*. We remark that this must be done before schema transformation takes place. The tuple representing l as a dimension of fact f must be deleted in relation *FactHasDim*. Finally, the tuple representing l in Ft_<f> is deleted from relation *Columns*.

figure 4-37: operation *delete dimension*$_L$

4.4.5. Putting Things Together: the complete Transformation Algorithm

As refinement to chapter 4.4.3, we present the second basic idea for the transformation algorithm: the definition of processing priorities for the logical evolution operations. More precisely, we define five groups of logical evolution operations. The transformation algorithm always tries in the first step to find an applicable operation sequence of the highest priority. If no such sequence is found (either because the preconditions do not hold (yet) or there is simply no such sequence in the evolution job), the algorithm tries to find an operation sequence of the next lower priority class and so on. If an applicable sequence has been found and processed, i.e. the sequence of conceptual evolution operations has been transformed to a logical evolution operation and removed from γ, the transformation algorithm tries to find again an applicable sequence of the highest priority (which may then be applicable because of the execution of the operation just processed).

There are five groups of logical evolution operations with different processing priorities. We will now explain why we need these priorities at all and why certain operations belong to a specific group. Furthermore, we will explain why we assigned a given priority to a group of operations opposed to other groups.

Basically, three ideas explain the priority schema in general. The first idea is that we always try to process operations dealing with attributes first (see figure 4-38 for the groups ordered according to their processing priorities). The reason for that is that an attribute operation always bears the necessary context for processing the operation. It is immediately clear which effects on which tables the attribute operation has because the attribute is always bound in its semantics to the fact or dimension level it belongs to. We could also say that although there is no clear difference on the ME/R graph layer, attributes are some kind of second-class citizens as opposed to facts and dimension levels. This is mainly because an attribute alone is not a valid part of an ME/R graph and also concerning an MD schema, an attribute alone is no meaningful part of the schema. The idea to process attribute operations first also saves us from defining additional logical evolution operations. For example, if we would not have the highest priority for these operations, we would also have to consider complex operations that insert or delete a fact together with a measure attribute or a dimension level together with an describing attribute.

The next idea of the priority scheme is to process applicable sequences of operations that are as long as possible. This idea explains why the operation `insert fact table with dimension table`$_L$ which is composed of three conceptual schema evolution operations has been assigned priority class 2. Similarly, the operations in priority class 3 are composed of two operations, whereas the operations in priority class 4 and 5 only consist of a single conceptual schema evolution operation.

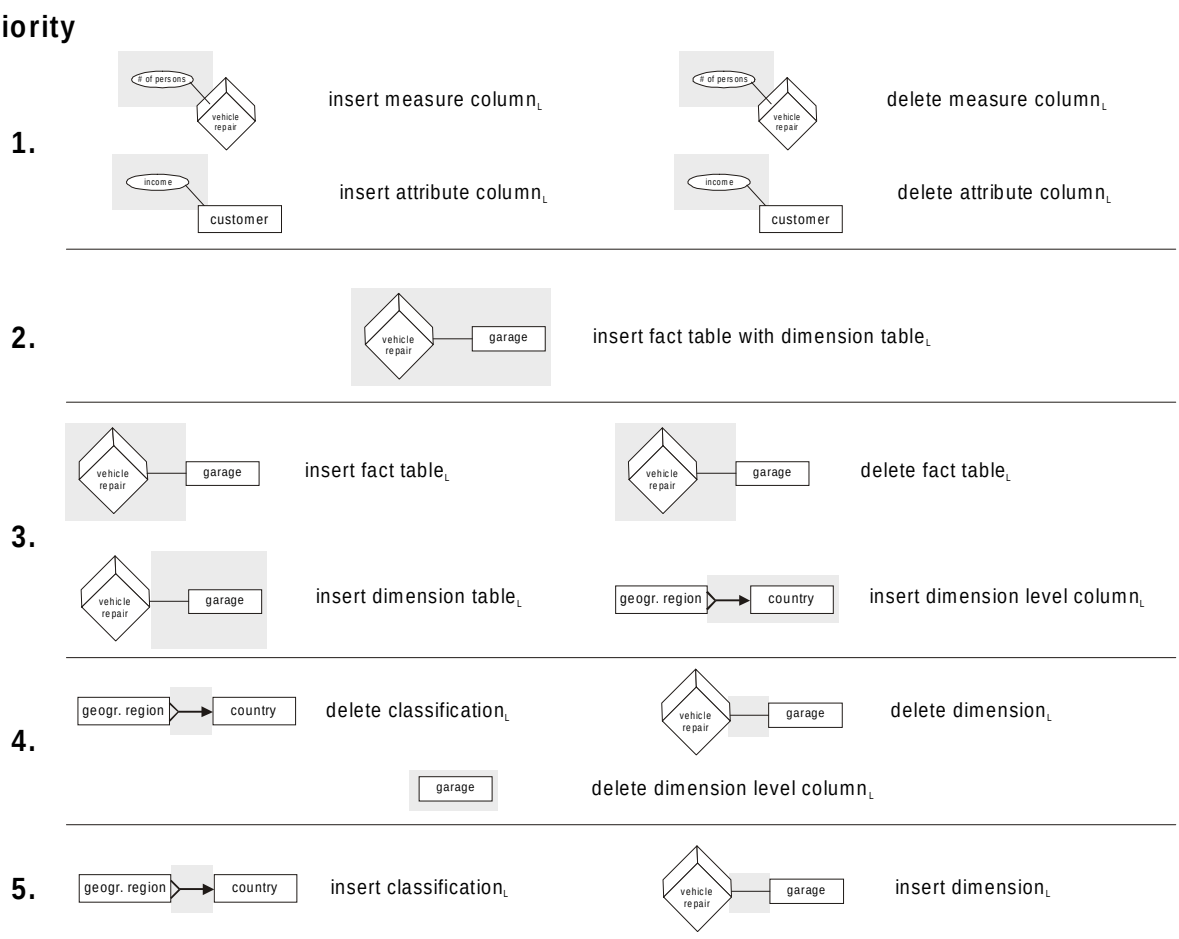


figure 4-38: processing priorities of logical evolution operations

The last idea for the understanding of the priority scheme explains the difference between the priority classes 4 and 5. We assume again our standard vehicle repair scenario as shown in figure 4-39 and assume an evolution job that breaks the classification hierarchy of the vehicle dimension and additionally inserts a new fact vehicle sales, see figure 4-40.

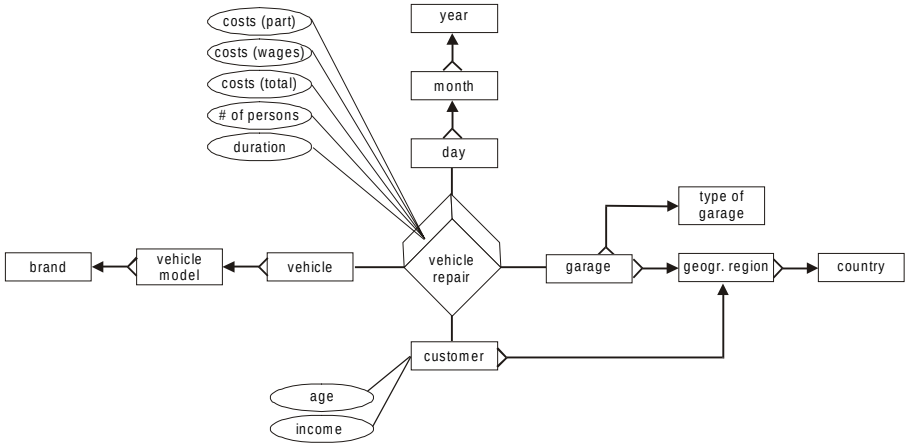


figure 4-39: vehicle repair example

γ

```
delete classification (vehicle, vehicle model);
insert fact (vehicle sales);
insert dimension level into fact
  (vehicle model, vehicle sales);
insert classification (vehicle, brand);
```

figure 4-40: example evolution job

If we regard the operations `delete classification (vehicle, vehicle model)` and `insert classification (vehicle, brand)`, we see that they are both applicable. The reason why we process the `delete classification` operation (priority 4) before the `insert classification` operation is a drawback of the transformation algorithm design. As sketched in chapters 4.4.3 and 4.4.4, we mark dimension levels for deletion when processing a `delete classification` operation. Consequently, when we process an insert classification operation that re-inserts part of the deleted classification hierarchy (dimension level brand in the example) we may merely reset the deletion mark. If we would not solve this problem by priorities, we would have to check for this case at the garbage collection which would make the algorithm more complicated than an additional priority class. An anlogous argumentation applies to the case of the `insert dimension into fact` operation.

The priority scheme together with the grouping of the operations assures the unique and complete parsing of the sequence γ.

```
/* input: sequence gamma of conceptual schema evolution operations
/* output: sequence lambda of logical evolution operations
transformation_algorithm (ordered sequence of operations gamma);
{
 /* search_priority: is increased (corresponds to lower priority) until
 /* operation sequence of the highest possible priority is found
 /* found: is set to TRUE if an applicable operation sequence has been found
 /* ops_positions: used to store the positions of matched component
 /* operations (maximum: 3 component operations) in gamma
 int search_priority := 1;
 bool found := FALSE;
 array ops_positions[MAXOPS];

 while not isempty (gamma) {  /* gamma is completely reduced
       /* search for an operation sequence of the highest possible priority
       /* for testing if sequence is applicable:
       /* always check the preconditions against the current state
       /* return the indices of the matching component operations
       search_in_γ_for_applicable_operation_sequence
                                        (search_priority, found, ops_positions);
      if (found)  /* if applicable sequence has been found
                  /* positions of matching component operations
                  /* have been stored in ops_positions
            then  {/* transform to logical evolution operations
                   transform_operations (ops_positions);
                   /* remove matching component operations from gamma
                   remove_operations_from_gamma (ops_positions);
                   /* start new search with highest priority
                   search_priority=1;
                   break;
                  }
            else  /* no operation sequence with current search_priority
                  /* found: select next search priority
                  {search_priority := search_priority + 1;
                   /* if no applicable operation of the lowest priority
                   /* has been found: sequence must be wrong
                   if search_priority == MAXPRIO+1 then error_handling();
                  }
      } /* while */

/* call garbage collection */
garbage_collection();
} /* transformation algorithm
```

figure 4-41: transformation algorithm

Next, we describe the complete transformation algorithm which is shown in figure 4-41. As mentioned in chapter 4.4.3, its input is the sequence γ of conceptual schema evolution operations. By parsing the algorithm transforms (i.e., rewrites) this sequence to a sequence of logical evolution operations as output. In the main loop, it searches in γ for applicable (w.r.t. to the preconditions[16]) sequences of conceptual schema evolution operations. If such an applicable is found, the subroutine *search_in_γ_for_applicable_operation_sequence* returns the index positions of the component operations in γ. These component operations are then transformed to a corresponding logical evolution operation and removed from the sequence γ. Then, the algorithm starts again the search for applicable operations of the highest priority group (reset search_priority to 1). Otherwise, if no applicable operation sequence of the current priority is found, the algorithm searches for an applicable sequence of the next lower priority class.

The algorithm terminates if the sequence γ is empty which corresponds to the complete derivability of each sequence γ. The logical evolution operations are complete in the sense that every such sequence of conceptual schema evolution operations can be parsed, i.e. transformed to a corresponding sequence of logical evolution operations. The uniqueness of the derivation is ensured by the design of the algorithm, more precisely: the priority scheme and the set of logical evolution operations and their composition.

In order to explain the processing model of the transformation algorithm, we extend and refine the example from chapter 4.4.3. We repeat the situation for the starting point: the vehicle repairs example in figure 4-42 and the evolution job in figure 4-43. The following text basically corresponds to the example in chapter 4.4.3, but has been extended by a detailed discussion about the priority scheme and the introduced logical evolution operations. These extensions correspond to the non-generic part of the transformation algorithm that have been designed for star schemas as schema template on the logical layer.

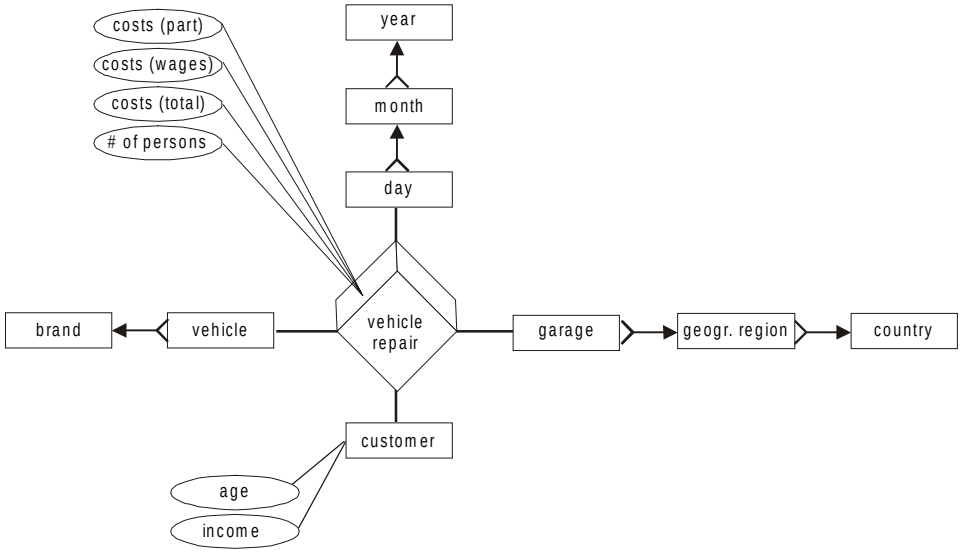


figure 4-42: example schema

[16] We remark that for the evaluation of the preconditions always the current state is checked against the preconditions.

γ

```
insert classification (customer,geogr. region);
insert fact (vehicle sales);
insert dimension level into fact
  (day, vehicle sales);
insert attribute (count);
connect attribute to fact (count, vehicle sales);
```

figure 4-43: example evolution job

As said before, the transformation algorithm tries to find an applicable operation sequence of the highest priority. The first operation `insert classification (customer, geogr. region)` is applicable (because both dimension levels are contained in the MD schema), but not of the highest priority class. The operation sequence `insert attribute (count); connect attribute to fact (count, vehicle sales)` belongs to the highest processing priority class, but is not applicable yet because the fact *vehicle sales* is not contained in the MD schema. Thus, the algorithm starts with a sequence of second priority class: `insert fact (vehicle sales); insert dimension level (day, vehicle sales)`. This sequence is applicable because the dimension level *day* already belongs to the MD schema.

The transformation algorithm processes this sequence, i.e. it transforms these two conceptual schema evolution operations to a logical evolution operation that transforms the star schema, adapts the instances and updates the meta schema contents. Finally, the two operations are deleted from γ.

Consequently, after this first step, we have the following intermediate situation: the resulting MD schema is depicted in figure 4-44 and the remaining evolution job γ is shown in figure 4-45:

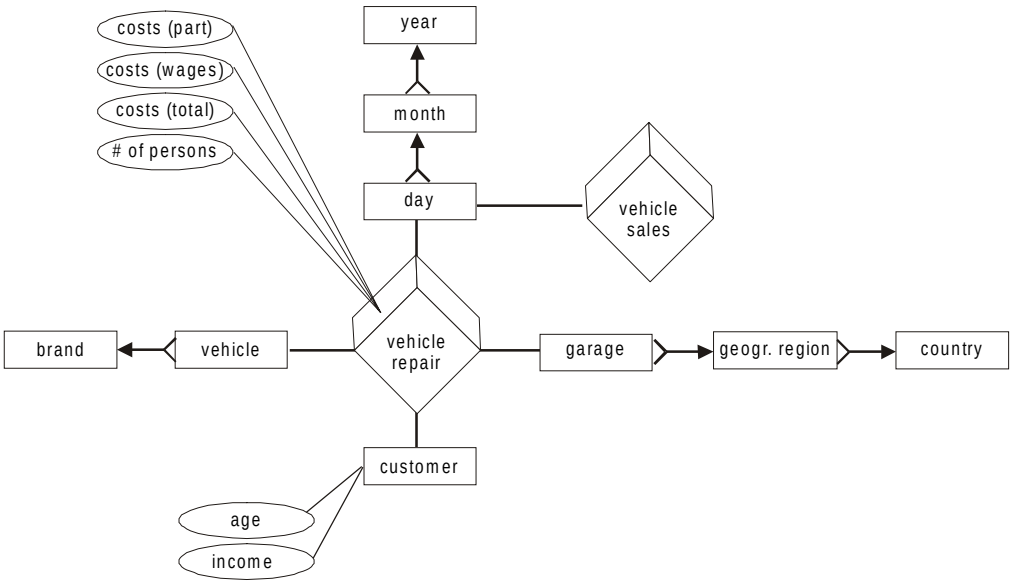


figure 4-44: example schema after the first transformation step

γ

```
insert classification (customer,geogr. region);
insert attribute (count);
connect attribute to fact (count, vehicle sales);
```

figure 4-45: example evolution job after the first transformation step

Now, as second step, both the sequences `insert classification (customer, geogr. region)` and `insert attribute (count); connect attribute to fact (count, vehicle sales)` are applicable, but the second belongs to the highest priority class and therefore is processed next. Finally, the operation `insert classification (customer, geogr. region)` is processed at the end of the transformation phase. The transformation algorithm holds because γ is empty.

We conclude the algorithm's description with a few words concerning the garbage collection. As carried out before, columns representing dimension levels are only marked for deletion in their corresponding dimension tables. Consequently, during the garbage collection phase, all marked dimension levels are deleted from their dimension tables with one exception. In order to present the exception, we repeat figure 4-18 as figure 4-46: the evolution job deletes the alternative path from the dimension levels *week* and *year*. This means that the column representing the dimension level *year* in the dimension table *Dt_time* would be marked for deletion. But, as there still exists another path from the base level *day* to the level *year*, the column representing *year* must not be deleted from *Dt_time*. The detection of this exception in the garbage collection algorithm is rather complex.

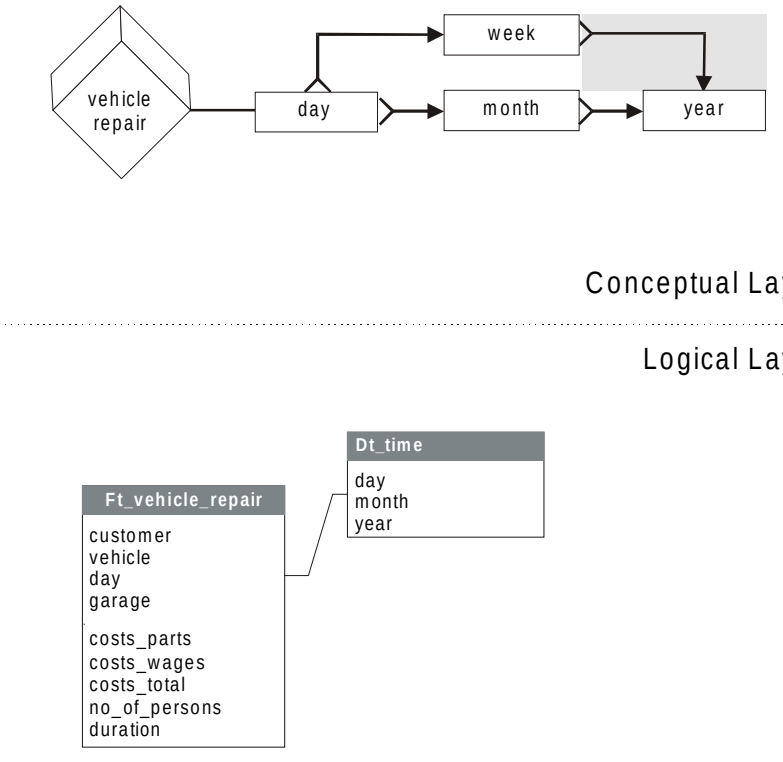


figure 4-46: deleting an alternative path

The handling for this situation can only be done at the end of the processing phase of the transformation algorithm and not during the processing of the `delete classification` operation because a matching `insert class` operation (that re-connects the level *year*) may be contained in another (distant) position in γ.

4.5. Summary

This chapter presented the formal mapping (and, consequently, the conncetion in the overall framework) between the conceptual layer (as described in chapter 3) and the database system layer (the logical layer in our layer model). Specifically, the detailed mapping information of our schema evolution algebra (MD schemas with schema evolution operations) serves as a complete formal specification which has been used as base for an implementation of the FI-ESTA framework.

Along with the detailed description of the mapping from a given MD schema to a relational star schema, we developed a meta schema as extension of the standard DBS system catalogue. This meta schema stores detailed information about the mapping. This information is necessary because most of the multidimensional semantics is hidden or even lost in the structure of a relational star schema due to the different expressiveness of an MD schema and a star schema. Especially, we showed how the meta schema reflects some powerful modeling capabilities of the ME/R approach, namely merging and shared dimensions. After an extended example, we refined this mapping description to a formal consistency criterion between the conceptual and logical layer.

Having introduced these prerequisites, we described how conceptual schema evolution operations are transformed to corrresponding logical evolution operations. Logical evolution operations transform the relational database schema, adapt the instances and update the contents of the meta schema accordingly. From a detailed discussion, taking into account the peculiarities of shared and merging dimensions, we derived several observations that led to requirements to the transformation algorithm. We discovered that local modifications of the ME/R graph may have effects on logical elements (i.e., tables and/or columns) representing other (i.e. graphically distant) parts of the ME/R graph. Additionally, we found that operations with the same semantics on the conceptual layer may lead to different semantics of operations on the logical layer. The main requirement and design idea derived from these considerations was to keep instance information as long as possible and therefore postpone the physical deletion of columns in dimension tables to a final garbage collection at the end of the transformation algorithm. Then, we described the generic part of the transformation algorithm wich is independent of the database schema template (e.g., star schema or snowflake schema) on the logical layer. Using an example, we sketched the processing phase of the algorithm. We explained how the algorithm identifies certain sequences of conceptual schema evolution operations which are then transformed to corresponding logical evolution operations.

Next, we presented the logical evolution operations for star schemas. We explained why we chose exactly these operations and showed some advantages of our design decisions. A detailed description of the semantics, parameters, and preconditions completed this chapter.

Finally, we put the pieces “algorithm” and “logical evolution operations” together and presented the refined complete transformation algorithm. We explained the priority scheme for processing the operations that ensures (together with the definition of the logical evolution operations) the complete and unique parsing of a given evolution job. We concluded with the explanation of the refined example and some key considerations about the garbage collection phase.

I would never die for my beliefs because I might be wrong.

(Bertrand Russel)

5. Discussion

Thoroughly, we have presented the FIESTA solution in detail in chapters 3 and 4. Now, we will discuss our approach. The discussion presented here is divided in three rather heterogenous parts:

As first step (chapter 5.1), we close the RDBS-generic part of the solution presented in chapter 4 and start with sketching the implementation of the overall BabelFish prototype. For this implementation, we have used several commercial products. We will show where and how the components of FIESTA have been embedded in the prototype.

Next, in chapter 5.2, we revisit the objectives for FIESTA (as introduced in chapter 3.3) and show how and to what degree we fulfilled them technically. We also discuss the solution of FIESTA in a wider context.

Finally, chapter 5.3 closes the last remaining open bracket and discusses related work (as presented in chapter 1.3). To that end, we evaluate where and how our approach differs from other solutions.

5.1. The FIESTA Implementation

The implementation of FIESTA is embedded in the implementation of the overall BabelFish prototype. Thus, we will introduce the components and interfaces of the BabelFish prototype and explain in detail the components of FIESTA.

An overview of the software components of the BabelFish system is shown in figure 5-1.

Since the underlying vision of BabelFish is to develop a repository-driven system for warehouse design and maintenance, the BabelFish repository constitutes the core of the prototype. The repository has been implemented using Softlab Enabler 2.0 [Sof98], a commercial repository product.

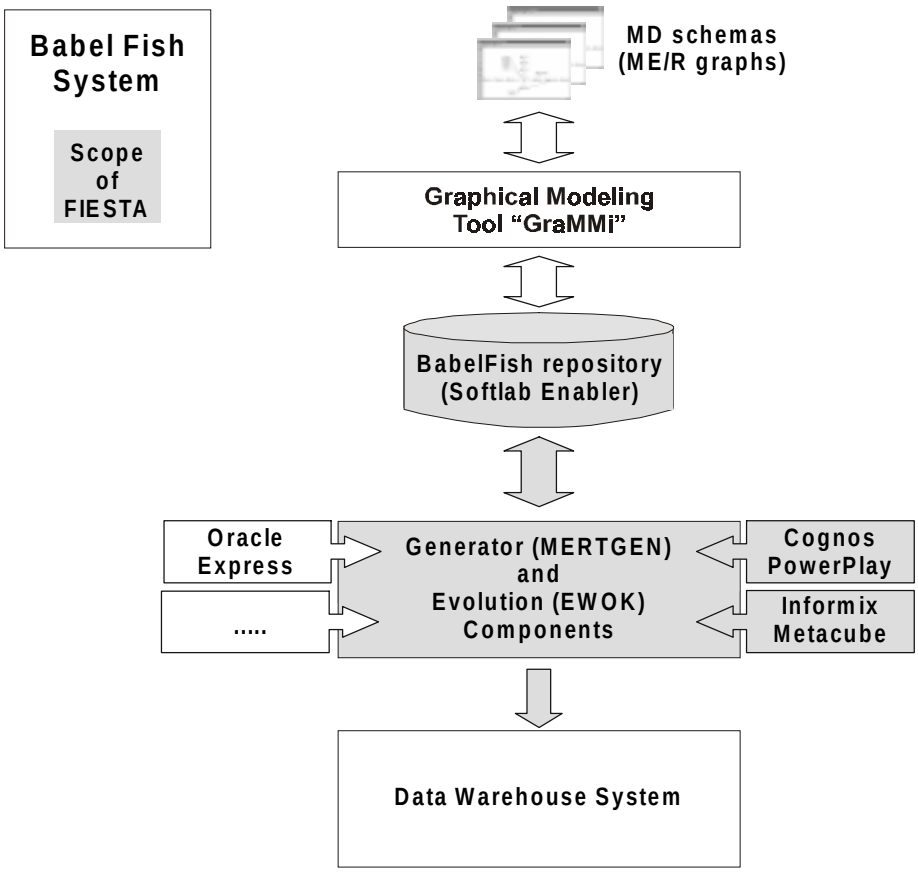


figure 5-1: Scope of the FIESTA implementation within the BabelFish prototype

As introduced in chapter 2.4.4, GraMMi ([Haa99], [SBH00]) is our graphical modeling tool for schema design and maintenance. When we developed the ME/R notation in the beginning of the BabelFish project, we expected to refine and maintain the notation during its evaluation in several commercial projects. Thus, we planned to involve end users (i.e., warehouse modelers) in the design of the ME/R notatation. This process implies frequent changes to the syntax and semantics of the modeling language. As a consequence, we required a flexible modeling tool that allows for a quick adaptation of the underlying modeling notation, without programming and recompilation. Therefore, we chose the meta-modeling approach that stores the modeling language as metadata in a repository. When change requests of the modeling notations arise, only the (meta) data in the repository has to be changed, but not the tool itself. The tool reads the modeling notation at startup from the repository. A sample screenshot of GraMMi is contained in figure 2-13, chapter 2.4.4.

The implementation of FIESTA is contained in two software components. Due to the complexity of the implementation and a wide range of requirements (as the examination of several commercial products) we decided to develop two components:

- a generator component, called MERTGEN [Hah00]. Objective of this component, carried out as a master thesis co-supervised by the author, was to use an ME/R graph in GraMMi and to transform it to corresponding logical schemas suited for two commercial products: Cognos PowerPlay and Informix MetaCube. Basically, MERTGEN reads an ME/R graph from the repository, transforms this graph according to the peculiarities of the target system and generates corresponding scripts that build a logical schema in the target system and update the tool meta schema.

- an evolution component, called EWOK [Vet99]. Objective of this component, carried out as an internship co-supervised by the author, was to transform schema evolution jobs to corresponding logical evolution jobs for Informix MetaCube. EWOK reads the evolution jobs from the repository and creates scripts that adapt the relational star schema in the Informix database and update the contents of the MetaCube meta schema accordingly.

We will now describe both components and issues (e.g., the meta schema) that are relevant for both components in detail.

The generator component MERTGEN uses a conceptual multidimensional schema, visualized by its ME/R graph, as input and generates by the use of the graph grammar (see chapter 2.4.3 and [Hah00]) corresponding scripts to generate logical database schemas for two commercial products: Cognos Powerplay ([Cog98a],[Cog98b],[Cog98c]) and Informix Dynamic Server (Version 7.30, [Inf98a], [Inf98b]) together with Informix MetaCube (Version 4.0.2, [Inf98c], [Inf98d]). MERTGEN has been developed using Microsoft Visual C++ version 5.0 under Windows NT.

The underlying research problem of MERTGEN, the transformation of a graphical model representation to a script language for the target system, is a compiler construction problem. When investigating the single phases of a typical compiler [ASU88], we note the following similarities and differences from a “classical” compiler to the transformation algorithm presented in [Hah00]: Starting point is not a source program, represented by a string, but a conceptual schema, represented by a ME/R graph. Thus, a scanner for the lexical analysis is replaced by reading the model representation (from a repository) and internally building the directed, acyclic graph representation. The syntax analysis phase checks for the correctness of the graph which represents the MD model using a graph grammar (see chapter 2.4.3). Details of the algorithm that checks the correctness by using the graph grammar are given in [Hah00]. A semantical analysis could check additional integrity constraints of the graph that cannot be represented by the graph grammar. Examples are cycles or multiple edges of the same type between the same nodes. After the analysis phase, the graph is transformed according to the possibilities and restrictions of the target system. The result of this phase is still a graph, but the graph is already adapted to the peculiarities of the target system. Consequently, this phase can be regarded as the phase generating intermediate code for a regular compiler. Optimizing the models for the target system requires information that cannot be contained in the ME/R model.

The code generation delivers commands that build the logical schema using the notation of the target system. The algorithm for generating these target system commands is strongly system-specific and cannot be generic due to the highly different expressiveness of commercial databases and tools. Not all features of the ME/R modeling notation can be directly expressed in all commercial tools. The support for “difficult” features of the ME/R notation in the examined products is shown in figure 5-2.

	Cognos Powerplay	**Informix Metacube**
describing attributes of dimension levels	not supported	supported
alternative paths	not supported	supported
merging dimensions	not supported	not supported
multiple facts	not supported	supported

figure 5-2: limitations of commercial OLAP products

Since some features of the ME/R notation were not directly expressible in the target systems, an information preserving solution had to be found. Basically, the ME/R graph is transformed according to the limitations of the target system. We refer to [Hah00] for details on the transformations.

Having transformed the ME/R graph accordingly, MERTGEN generates scripts that build the logical schema and fill the contents of the tool's meta schema. These scripts are SQL scripts in the case of Informix MetaCube and a proprietary script language called MDL in the case of PowerPlay.

The evolution component EWOK [Vet99] assumes an existing conceptual schema evolution job and transforms it to corresponding SQL commands that transform the logical schema in the relational database and update the contents of the MetaCube meta schema accordingly. Common result of MERTGEN and EWOK is a precise integration of the FIESTA meta schema with the tool specific meta schema of Informix MetaCube. The different meta relations of MetaCube and their counterparts of the ME/R notation are summarized in figure 5-3.

Meta Table Name in MetaCube tool	**Corresponding Element in the ME/R Notation**	**Description**
Dss_system	complete model	a conceptual OLAP model
Fact_table	fact node	data about all facts of the model
Ui_fact_table	-	user interface configuration of the facts
Dim	dimension	data about all dimensions of the model
Ui_dim	-	user interface configuration of the dimensions
Fact_dim_mapping	connects edge between a fact and its base levels	assigns dimensions to facts
Ui_fact_dim	-	according user interface configuration
Fact	measure	data about measures (attributes) of facts
Ui_fact	-	according user interface configuration
Dim_el	dimension level node	data about all dimension levels of the model
Att	attribute of a dimension level	describing attributes of dimension levels
Ui_att	-	according user interface configuration
Rollup	classification edge	representation of the classification relationship of two dimension levels
Dss_sequence	-	counter for system variables
Dss_string	-	string variables for internal maintenance (e.g. SQL scripts for assigning a level value to dimension levels)

figure 5-3: Informix MetaCube metadata tables

We present this tool specific meta schema as an example for a meta schema. The scope of meta data in this meta schema has influenced the design of the FIESTA meta schema (described in chapter 4.1.5) which constitutes a generalization of tool-specific meta schemas.

EWOK is a tool component developed in Java with JDK version 1.2. It reads evolution jobs as input from the repository (for details of the meta schema representing evolution jobs, see [Vet99]) and transforms this sequence of conceptual schema evolution operations to SQL scipts that transform the relational star schema and update the contents of the meta schema. To that end, EWOK works with the following metadata contained in the BabelFish repository:

- the conceptual schema, represented as ME/R graph (see [Haa99] for details of the relevant part of the meta schema)
- the logical database schema (i.e., the structure of the star schema) representing the conceptual schema on the database layer,
- the evolution jobs as sequence of schema evolution operations (with pointers to the relevant parts of the conceptual MD schema), and
- the relationships between the repository objects (e.g., from a node of the ME/R graph to the corresponding element in the star schema)

During the parallel implementation of the three main components GraMMi [Haa99], MERTGEN [Hah00], and EWOK [Vet99] (see figure 5-1 for the components), the FIESTA repository schema was continously extended and integrated to cover this range of metadata. Thus, the resulting overall repository schema is a joint result of these works.

5.2. Conformity of the FIESTA solution with its objectives

As central part of the discussion of the solution developed in this thesis, we revisit the objectives that we have introduced in chapter 3.3. For each objective, we evaluate to what degree we have achieved it and sketch the main technical ideas having contributed to its fulfillment.

- support of the full design and maintenance cycle: FIESTA covers all design phases of the design and maintenance cycle (figure 1-2) and supports the implementation phase. FIESTA is usable both for initial schema design and the adaptation of a system populated with data. This objective has been completely fulfilled on the conceptual and the logical layer. The design of the conceptual schema evolution operations guarantees both the creation of MD schemas from scratch (in this case, the parts of the multidimensional data model representing instances are empty) and the modification of existing schemas during maintenance. As an advantage of our design, the modified parts of a given multidimensional schema are clearly identifiable. Additionally, the design of the logical evolution operations together with the transformation algorithm guarantees this support on the database system layer. This is particularily reflected in the design of some logical evolution operations that are basically used for either creating a new star schema (e.g., the `operation insert fact table with dimension table`$_L$, creating the minimal meaningful entity on the logical database layer) or assume an already existing star schema and modify it accordingly (e.g., the `operation insert measure column`$_L$, extending an already existing fact table).

- formal definition of semantics of evolution operations: this objective has also been completely fulfilled on the conceptual and the logical layer. On the conceptual layer, the se-

mantics of the schema evolution operations are precisely defined by the definition of the schema transformation and the instance adaptation in terms of the multidimensional data model. Operations offering alternatives concerning the semantics (e.g., cascading vs. non-cascading deletes) are expressible by different compositions of fine-grained operations. On the logical layer, the semantics of the logical evolution operations are specified in terms of transformations of the database schema, adaptations of the relational instances, and updates of the meta schema contents. Additionally, the transformation algorithm together with the priority schema guarantees a clearly defined transformation between the layers and execution of the resulting logical evolution job. Existing instances are kept as long as necessary in order to avoid loss of instance information and the final garbage collection phase cares for the removal of superfluous data.

- definition of fine-grained schema evolution operations: this objective has not only been completely fulfilled, but was also extremely helpful for the solution. The fine-grained definition on the conceptual layer allows for different variants of the graphical notation (e.g., assuming attributes as second-class entities and representing them graphically as parts of the dimension level or fact icon) and guarantees precise and flexible semantics of the evolution jobs. However, beyond these advantages, the fine-grained approach was also extremely beneficial for the transformation to logical evolution operations. Only the fine-grained approach of the conceptual schema evolution operations allows for the correct composition in the definition of the logical evolution operations. It further allows an easy definition of logical evolution operations for different constructs on the logical layer (e.g., a snowflake schema instead of the star schema).

- automatic adaptation of logical schema, instances and tool metadata: the precise and formal mapping between the conceptual and logical layer constitutes the necessary prerequisite for this objective. Again, FIESTA fulfills this objective completely by the mapping and the transformation algorithm with the corresponding logical evolution operations. The logical evolution operations transform the logical database schema, adapt existing instances and update the contents of the tool metadata (represented by the FIESTA meta schema). The alternatives of an immediate adaptation of the instances vs. filter for access to the instances seemed not beneficial for OLAP systems. This approach yielded valuable results in the area of object-oriented schema evolution, but not for OLAP systems. Thus, we omitted the issue.

- formulation and check of integrity constraints: integrity constraints that ensure consistency play a vital role in the FIESTA approach. Therefore, we formally defined all arising notions of consistency: consistency of multidimensional schemas, correctness criteria for ME/R graphs and evolution jobs, and the mapping between the conceptual and logical layer as a consistency criterion between the two layers (see also the FIESTA schema evolution problem shown in figure 3-5). Currently, we have no means to express further user-defined constraints because we found no application beyond the defined notion of consistency.

- use of a repository system for all meta data: the meta schema plays a central role in the design of FIESTA. It is generic for RDBS on the logical layer, but its design has been influenced by experiences with commercial tools. In the prototype implementation, we used the commercial product Enabler [Sof98] by Softlab Corporation.

So far, we have discussed the degree to which FIESTA has fulfilled its objectives. We remark that the developed solution is fully compliant with the main objectives.

We conclude this chapter by a slightly philosophical discussion of FIESTA in a wider context. As pointed out in chapter 3.1, the roadmap to schema evolution is generic, or precisely: data model dependent. FIESTA has been specifically developed as framework for the design and evolution of multidimensional OLAP systems that are implemented using a relational DBS. However, since FIESTA is only a specific instantiation of the generic schema evolution roadmap, the "schema" of FIESTA could also be transferred to other similar scenarios dealing with schema evolution as part of the schema design and maintenance cycle. Only specific parts of the schema evolution roadmap would have to be exchanged.

If we exchange the ME/R notation with a different graphical notation for MD schemas, the correspondance of the graphical notation with our MD data model would have to be adpated. All other parts of our solution would still apply. Similarly, if we would choose to use snowflake schemas instead of star schemas on the logical layer, the logical evolution operations with their priority schema and the design of the mapping information in the meta schema would have to be adapted.

A transfer of FIESTA to the area of object-oriented schema evolution research is also possible. To that end, all "instances" of FIESTA would have to be modified, but the general framework would still hold: a formal object-oriented data model with schema evolution operations and a graphical representation of object-oriented schemas would have to be chosen (or developed), the mappings between schemas and their graphical representation together with the mapping to the logical layer would have to be adapted. Similarly, corresponding logical evolution operations (including their processing priorites) would have to be adapted. We think that the logical evolution operations for the case of object-oriented schema evolution are much simpler than for the multidimensional case. The result of this transfer would be a tool-supported environment for object-oriented schema design and maintenance.

5.3. Related Work

This chapter compares the FIESTA solution with related work. We refer back to chapter 1.3.5 which compared the objective vision of FIESTA (chapter 1.2) with the current state of the art (chapters 1.3.1 to 1.3.4). As already shown there, only evolution approaches that are both purely conceptual and based on a multidimensional data model can be directly compared with FIESTA. Thus, the approaches of Bellahsene [Bel98], Mohania / Dong ([MD96], [Moh97]), Rundensteiner et al. [RLN97], and Quix[17] [Qui99] have rather different overall objectives which make them incomparable to the FIESTA solution.

5.3.1. Multidimensional Data Models

The MD data models presented in the literature (chapter 1.3.4) provide no direct schema evolution support. Nevertheless, they have influenced our work. Since the overall starting point of this thesis was the evolution algebra, we began with a comparison of existing formalizations of the MD data model ([BSHD98], [SBH99]) in order to develop an own formal model which should be especially suited to express schema evolution operations (which were then published in [BSH99]). The deep study of the single constructs formalizing the MD data model has therefore influenced our MD data model as introduced in chapter 3.5. Basically, our model is influenced by the formalizations of Cabbibo / Torlone [CT98] and Vassiliadis [Vas98] because

[17] The solution of Quix is purely conceptual, but not fully based on a multidimensional data model. Although the approach introduces schema evolution operations, it only discusses their effects on quality factors and not on the data model.

these approaches are completely conceptual and multidimensional due to their cube-oriented view.

In contrast to all other formalizations of the MD data model, we do not focus on OLAP operations, but on schema evolution operations. This may explain some differences in our formalization compared to others, especially our clear separation of schema and instances as separate parts of the formal MD data model.

Since OLAP systems are commonly implemented on top of relational DBS, the mapping of MD schemas to relational structures has to be formally defined. The approach of Gyssens and Lakshmanan [GL97] for example defines a dualism between the tabular representation of multidimensional tables and their corresponding relations. We prefer having a purely cube-oriented view on the multidimensional layer and thus provide a purely conceptual data model. We also contribute a formal mapping of MD schemas to relational star schemas (chapter 4.1).

We remark that the use of relational database systems (RDBS) on the logical layer is not the only possible architecture for OLAP systems. Some commercial systems (e.g. Oracle Express [Ora97]) provide also a multidimensional model on the logical layer. For the use of such a multidimensional database system (MDDBS), the mapping defined in chapter 4.1 would not include the shift of the data model (from multidimensional to relational) and, as a consequence, the mapping would become less complex (but still, it would be far from being trivial). However, since the schema evolution capabilities of today's MDDBS are poor and RDBS offer at least basic constructs for schema evolution, we decided to use RDBS on the logical layer.

5.3.2. Graphical Modeling Notations for Warehouse Design

Another research area that is only partially relevant as state of the art for FIESTA are graphical modeling notations for the design of warehouse schemas. Although being a neglected issue in the beginning of data warehouse research, the issue has received growing attention lately (see e.g. the newer approaches [LST99] and [MCA+00]).

FIESTA uses the ME/R notation ([SBHD98] and chapter 2.4) that has been developed as part of the BabelFish project in which this thesis is embedded. Since ME/R models represent MD schemas graphically, a formal dualism between both representations has been defined (see chapter 3.6). The ME/R modeling notation has been designed in close cooperation with our MD data model in order to provide a clear and intuitive dualism. However, some non-trivial problems still remain (e.g., redundant edges in ME/R graphs) that had to be addressed. Due to the generality and modularity of our framework, a different graphical modeling notation (e.g. the DF notation, see below) could only be used. In this case, only the mapping between this representation and the MD data model would have to be adapted.

Golfarelli et al. ([GMR98], [GR98]) proposed a methodological framework for data warehouse design based on a conceptual model called dimensional fact (DF) scheme. They introduce a graphical notation and a methodology to derive a DF model from E/R models of the data sources. Having characterizing a workload in terms of data volumes and expected queries, their methodology can be used for the logical and physical design of warehousing systems. As it can be seen, this methodology complements our work. However, we argue that the proper starting point for schema design should be a clearly conceptual model as the warehouse modeler sees his universe of discourse. Thus, the information available at the sources can be complementary information, but should not limit the scope of the conceptual schema. Although the DF modeling technique supports semantically rich concepts, it is not based on a formal data

model. Furthermore, the framework does not concentrate on evolution issues which we believe is an important feature for the design and maintenance cycle.

The Multidimensional Modeling Language (MML), introduced in [Har99], is an object-oriented and extendable conceptual multidimensional modeling language. Extendable refers to the design principle that MML is based on meta models. As a consequence, MML provides no own graphical modeling notation. In [Har99], two exemplary graphical notations are introduced: MML* and mUML (multidimensional UML). mUML ([Har99], [HH99], [Her00]) is a multidimensional extension of the Unified Modeling Language (UML, [Rati97]) for the design of conceptual multidimensional schemas using the UML notation. Main reason for this design decision is the modeling support offered by existing commercial CASE tools (e.g. Rational Rose [Rati98]) and its inherent extensibility. mUML can be seen as the object-oriented counterpart to our ME/R notation, extending the UML instead of the E/R notation for multidimensional schema design. The overall approach for MML and mUML seems to have followed our idea of conceptual warehouse design and our layer model, but our own approach provides no formal meta model (the counterpart to MML would be the meta model of the ME/R notation which has not been formally designed in the BabelFish project). Support for schema evolution in MML is provided only by validity stamps of the elements of a schema. There are no schema evolution operations provided. Although a formal mapping from MML to relational star schemas is defined in [Har99], it is left open how the validity stamps can be exploited for transforming existing relational schemas under evolution.

5.3.3. Approach of Chamoni and Stock

The approach of Chamoni and Stock has been introduced in chapter 1.3.3.4. When comparing this approach to FIESTA, the following basic differences can be identified:

The approach of Chamoni and Stock concentrates on changes in the classification hierarchy of dimensions (like the initial approach of Kimball [Kim96b]). Here, the evolution of classifications of single dimension elements (e.g., a product) according to the next hierarchy level (e.g., product group) is regarded. Thus, it is possible to reflect the evolution of classification hierarchies over time, but the approach does not consider modifications of the structure of the dimension hierarchy (e.g., inserting a new dimension level). As a consequence, the approach does not provide schema evolution operations. A mapping to a possible database implementation is also missing. Summarizing, we may say that the two approaches are only loosely related, but may be complementary because FIESTA in its current state does not handle modifications of instances.

5.3.4. Approach of Hurtado et al.

The approach of Hurtado et al. ([HMV99a], [HMV99b]) is certainly the approach that comes closest to the FIESTA solution. Thus, we will discuss what the approaches have in common and where they differ.

Both FIESTA and the approach of Hurtado et al. introduce schema evolution operations. The following schema modification operations are proposed in [HMV99a]:

(1) Generalize: this operator creates a new level, l_{new}, to which a pre-existent one, l, rolls up. A function f must be defined from the set of instances of l to the domain of the new level. This function contains the classification information for the two levels.

(2) Specialize: this operator adds a new level l_{new} to a dimension. Level l_{new} will roll up to the lowest level of the dimension, becoming the new lowest level. Again, a function f must be

defined for classifying the instances of l_{new} according to the instances of the next higher level.

(3) Relate: the relate operator defines a roll up function between two independent (i.e. related by direct or in-direct classifications yet) levels belonging to the same dimension. As a condition, a function must exist between the instance sets of the two levels being related, such that the dimension instance remains consistent. Especially, all redundant roll up functions which may appear by applying the operation must be removed.

(4) Unrelate: the unrelate operator deletes an existing roll up relation between two levels. The execution of the operator must guarantee that levels below and above the two levels still are reachable. I.e., necessary roll up functions that extend over the two levels to be unrelated must be defined implicitly.

(5) DeleteLevel: this operator deletes a level and its roll up functions. The level to be deleted cannot be the lowest in a dimension hierarchy (unless it rolls up only to one higher level). Again, the roll ups between levels above and below the level to be deleted must be defined implicitly to ensure consistency.

The first paper [HMV99a] defines two instance update operators which have been extended by four complex instance update operators in the second paper [HMV99b] (but no more schema modification operators). The complete set of instance update operations consists of the following operations:

(1) Add Instance: inserts a new element into a level. The operator must be provided with the roll up classification for the new element, i.e. the set of elements of the next higher dimension level this new element rolls up to.

(2) Delete Instance: this operator deletes an element of a dimension level. It may only be applied if no other element of a lower level rolls up to this element.

(3) Reclassify: is a complex operation defined by a sequence of delete instance and add instance operations. Certain conditions are named to ensure the consistency within the dimension.

(4) Split: splits a dimension element to a set of new dimension elements (i.e. including the roll ups)

(5) Merge: the inverse operation to split, i.e. combines a set of dimension elements to a single one.

(6) Update: this operator just changes the value of an element, keeping the structure and the roll up functions unchanged.

When considering the applicability (defined by the pre-conditions) of the operations, we note that there is a strong notion of consistency between the instances of a dimension, especially w.r.t. the classification relationships among them.

The papers define two different mappings using a relational database: a star schema (denormalized) and a snowflake schema (normalized) approach. Both transformations are formally defined. The complexity of the operations' algorithms are discussed for both possible transformations (without a detailed cost model).

In order to maintain the implemented data cube, specific algorithms are given. Main task of these algorithms is to adapt the pre-calculated aggregations. Wherever this is not possible due to structural changes of the data cube (operations DeleteLevel and Specialize), a new base fact table is defined. The proposed algorithm for incremental maintenance is an extension of the

summary delta method, proposed by Mumick et al. [MQM97]. Since the aggregations can be computed by using other already computed aggregations, the view lattice approach of Harinarayan et al. [HRU96] for maintenance is adapted accordingly.

In contrast to FIESTA, the schema evolution operations in [HMV99a] focus only on changes in the dimensions and their hierarchies. We provide a set of schema evolution operations that comprise modifications on any part of the cube structure (including describing attributes of dimension levels and measures) and do not restrict the operations only to modifications of the dimensions. Thus, concerning schema evolution operations, the FIESTA framework provides a superset of the operations given in [HMV99a] meaning that all operations defined there can be expressed by FIESTA's schema evolution operations (but not vice versa). Next, [HMV99a] limits insertions of levels to certain positions (lowest and highest) in a dimension hierarchy. Our framework allows random insertions of dimension levels at any place of a given dimension hierarchy. A clear strength of the approach of Hurtado et al. are the operations dealing with the evolution of instances, i.e., changes in the classification hierarchy of instances. Here, a combination of this work with FIESTA seems promising. We plan to extend FIESTA by a comprehensive set of instance operations (see also chapter 6) that refine our framework also to e.g. the load phases of an OLAP database. As said before, the approach of Hurtado is weak when it comes to modifications of the facts and especially the fact tables. These are only regarded as part of the instance adaptation (thus, facts are not regarded at all on a conceptual layer). When modifications of the dimensions require an adaptation of the fact table, a new fact table is created. No instance adaptation for existing instances is formally described.

The design of fine-grained evolution operations and the composition to complex operations is common to both approaches.

A combination of the two complementary approaches seems promising. An issue where the approach of Hurtado complements FIESTA are the instance operations. Currently, FIESTA regards to insertions or deletions of instances and assumes this task is performed outside the framework. [HMV99a] introduces dedicated instance operations and discusses their effects (e.g., if a store is deleted, what are the resulting necessary modifications of the fact table instances?).

5.3.5. Work in progress

A recent approach to data warehouse modeling is [MCA+00]. The paper introduces IDEA-DWCASE, a client-server tool that supports a data warehouse construction methodology (called EINSTEIN) and automates the process of generating multidimensional database schemata. Unfortunately, the publication references only a software demonstration, no further literature is currently available. Especially, it is unclear whether schema evolution is supported by the proposed framework.

Another recent approach is the TEMPS approach of Günzel [Gün00]. TEMPS (see also chapter 1.3.3.5) aims at providing time information for schema versioning. The approach is based on a multidimensional data model. The proposed set of schema evolution operations seems being influenced by the FIESTA operations. The overall vision of TEMPS promises a very powerful approach where not only certain versions of the schema, but also of the instances and combinations thereof can be combined. Thus, the overall vision resembles a combination of the approach of Chamoni and Stock with FIESTA. Since the work is still at an early state, an in-depth comparison is not possible yet.

You see things and say 'Why?' but I dream
things that never were and say 'Why not?'
(George Bernard Shaw)

6. Conclusions and Future Work

Application areas for databases like data warehousing and OLAP use the multidimensional data model in order to describe the warehouse modeler's universe of discourse. OLAP applications typically assume a conceptual multidimensional schema to adequately reflect the application semantics.

In this thesis, we have introduced FIESTA, a methodology for the evolution of conceptual multidimensional schemas. Since schema evolution is not a completely new research issue but has been discussed for relational database systems and received considerable attention due to the complexity of the problem in object-oriented database systems, we have defined a generic roadmap to schema evolution. It consists of an evolution algebra (i.e., in general, a conceptual data model together with schema evolution operations defined on it), an execution model (i.e., propagation rules and integrity constraints), and – as refinement of the execution model – a software architecture. FIESTA is a specific instance of this roadmap, applied to the multidimensional data model.

The main research contributions of FIESTA can be summarized as follows:

- A formalization of the multidimensional data model that is purely conceptual (i.e., not assuming any implementation details) and puts a strong notion at the differences between multidimensional schemas and instances.
- A graphical representation of multidimensional schemas: the ME/R notation, an extension of the well-known and well-researched Entity Relationship modeling technique, especially designed for modeling multidimensional schemas.
- Since our vision is a graphical tool-supported environment for the design and maintenance of multidimensional schemas, we contributed a formal dualism that allows to use both the algebraic and graphical representation of a given multidimensional schema equivalently. To that end, we introduced a normal form for ME/R graphs and presented formal mappings between both representations.
- A set of fourteen conceptual schema evolution operations. These schema evolution operations together with the formal multidimensional data model constitute the core of our multidimensional schema evolution algebra. The proposed operations of FIESTA were the first

schema evolution operations that have been specifically designed for use with a multidimensional data model. Thus, our schema evolution operations constitute the starting point for research on multidimensional schema evolution which becomes a lively research area. Our proposal has also influenced and inspired new approaches like TEMPS [Gün00] and ODAWA [HH99], [Her00].

- In order to process the evolution of a given multidimensional schema in an underlying relational database system, we have developed a complete execution model for our conceptual schema evolution operations on the logical processing layer. To that end, we have defined a formal mapping between multidimensional schemas and relational star schemas. A dedicated meta schema stores the mapping information and keeps the multidimensional semantics that would be lost otherwise during the transformation to the semantically poor star schema. We refined this mapping to a consistency criterion between the conceptual multidimensional and the logical layer. Next, we presented a transformation algorithm that transforms a sequence of conceptual schema evolution operations to a sequence of corresponding logical evolution operations. These logical evolution operations transform the relational database schema, adapt existing instances, and update the contents of the meta schema. Core of the transformation algorithm is the set of fourteen logical evolution operations and a priority schema for their appliance.

We discussed our solution by presenting the implementation of FIESTA, which is embedded in the prototype of the BabelFish project for data warehouse design and maintenance.

A thorough and in-depth presentation of the state of the art covering a wide range of related research literature together with a broad and detailed discussion of the FIESTA solution with related approaches evaluated our solution. FIESTA fills the gap that other approaches leave open by the clear separation between the conceptual multidimensional and the logical relational layers, the complete and closed schema evolution algebra and the automatic adaptation of schema and instances.

When developing a framework like FIESTA, one always has to set a certain scope for the solution of the underlying research problem. Having finished a thesis, one always aims at relaxing this scope and transferring the developed solution to a broader focus. In the future, we would like to extend our framework by the following issues:

- first of all, although the conceptual part of FIESTA is generic and not specific for any implementation decisions, the processing of the evolution operations on the logical layer had to assume certain templates for the logical schema. We chose the star schema because it is the far most used schema template for implementing OLAP systems. As a drawback of this implementation decision, both the set of logical evolution operations and their priority schema is specific for star schemas. Since snowflake schemas are an alternative template for OLAP schemas in a relational database system, we will extend FIESTA by the corresponding logical evolution capabilities for snowflake schemas.
- due to the impedance mismatch between the ME/R notation and star schemas, some multidimensional schema entities have to be duplicated in star schemas. For example, a shared dimension level must be represented by two columns with the same name in different dimension tables. Although this problem is fully covered by the data in our meta schema, an extension (or automation) of the maintenance of the duplicated elements would be helpful. This could be done by triggers or integrity constraints that keep track of all instances in these duplicated attributes in order to avoid inconsistencies.

- our transformation algorithm always identifies applicable sequences of conceptual schema evolution operations and transforms these operations to corresponding logical evolution operations. Thus, it reduces the sequence of conceptual evolution operations step-by-step and generated SQL scripts for processing the operations on the logical layer. Currently, we have no formal framework for concurrency and transactions on this layer. In general, the scope of transactions should be on the granularity of a schema evolution job and not on applicable parts of jobs.
- currently, we assume that data is updated in the warehouse database using correct load algorithms that reflect the current state of the multidimensional schema. When e.g. a new dimension level is inserted, we assume that data insertion takes place outside the scope of FIESTA. An extension of FIESTA with respect to the load process of a data warehouse seems necessary for real world implementation.
- in order to increase performance, pre-aggregation is a common strategy for data warehouses. In this case, redundant aggregated data is stored in addition to the data warehouse database. This data has to be maintained which leads commonly to a view maintenance problem. But since not only the data may change but also the structure of the multidimensional schema, an extension of FIESTA to additional pre-aggregation tables seems necessary. This extension would complement the wide range of results available in the area of view maintenance for data warehouses.
- the logical evolution operations transform the logical schema. The SQL DDL scripts that are generated by the transformation algorithm may contain parts that refer to the same part of the relational star schema. For example, it is possible that a new table is created as result of the transformation of one conceptual schema evolution operation and then, as result of another conceptual schema evolution operation, the same table is modified later in the DDL script. Here, sophisticated optimizers that enhance the sequence of SQL DDL commands would be helpful.
- finally, the combination of FIESTA and techniques dealing with the physical design and optimization of star schemas seem a promising research area. So far, we have concentrated on the conceptual and logical layers and skipped the impacts of schema evolution to physical design issues like clustering or indexing. A good starting point for merging the results seems the Multidimensional Hierarchical Clustering (MHC) approach of Markl et al. ([Mar99], [MRB99]). Here, an order preserving encoding of hierarchies by surrogates is introduced which enables clustering of data with respect to multiple hierarchical dimensions. MHC can be implemented with any multidimensional access method, e.g. the UB-Tree ([Bay96], [Bay97]). An investigation of the impacts of schema evolution on clustering strategies seems worth further attention.

Appendix A: MD Schema Evolution Operations

This appendix presents the conceptual multidimensional schema evolution operations (see also chapter 3.7).

1. insert level

Criterion	Description
name of the operation	**insert_level**
informal explanation of semantics	inserts a new, isolated dimension level. The operation extends the set of levels without changing the classification relationships, thus creating an isolated element. Classifications relationships have to be defined separately.
syntax with input and output parameters	insert_level ($\mathcal{M}, \mathfrak{I}_m, l$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, new level name l **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$l \notin L$
post-condition(s)	$l \in L$, $\mathfrak{I}'_{m'} = \mathfrak{I}_m$
example	insert_level ($\mathcal{M}, \mathfrak{I}_m$, "brand")
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}' = <F, L' := L \cup \{ l_{new} \}, A, gran', class', attr'>$ *gran'*: $F \rightarrow 2^{L'}$; gran'(f) := gran(f) *class'* $\subseteq L' \times L'$; $(l_1,l_2) \in class' :\Leftrightarrow (l_1,l_2) \in class \ \forall l_1,l_2 \in L'$ *attr'*: $A \rightarrow F \cup L' \cup \{\perp\}$; *attr'*(a) := *attr*(a) **Instances:** No effects on instances because the operation inserts a new and empty dimension level without instances. $\mathfrak{I}'_{m'}$ = <R-UP, C, AV>

figure A-1: operation *insert level*

2. delete level

Criterion	Description
name of the operation	**delete_level**
informal explanation of semantics	deletes an isolated dimension level. The operation removes the level to be deleted from the set of levels. The level may not be connected to any other elements by classification or attribute relationships.
syntax with input and output parameters	delete _level ($\mathcal{M}, \mathfrak{I}_m, l_{del}$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, level name l_{del} of the level to be deleted **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$l_{del} \in L$, $l_{del} \notin gran(f)\ \forall f \in F$, $((l_{del}, l) \notin class \wedge (l, l_{del}) \notin class\ \forall l \in L')$, $attr(a) \neq l_{del}\ \forall a \in A$
post-condition(s)	$l_{del} \notin L$, $\mathfrak{I}'_{m'} = \mathfrak{I}_m$
example	delete_level ($\mathcal{M}, \mathfrak{I}_m$, "brand")
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}' = <F, L', A, gran', class', attr'>$. L' := L \ { l_{del} } *gran'*:= *gran* *class'* := *class* *attr'* := *attr* **Instances:** no effect because dimension members are deleted automatically. Thus: $\mathfrak{I}'_{m'}$ = <R-UP, C, AV>

figure A-2: operation *delete level*

3. insert attribute

Criterion	Description
name of the operation	**insert_attribute**
informal explanation of semantics	creates a new attribute without attaching it to a dimension level or fact. The operation inserts the new attribute in the set of attributes.
syntax with input and output parameters	insert_attribute ($\mathcal{M}, \mathfrak{I}_m, a_{new}$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, attribute a_{new} with dom(a_{new}) to be inserted **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$a_{new} \notin A$
post-condition(s)	$a_{new} \in A$, $\mathfrak{I}'_{m'} = \mathfrak{I}_m$
example	insert_attribute ($\mathcal{M}, \mathfrak{I}_m$, "age")
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}' = \langle F, L, A', gran, class, attr' \rangle$ $A' := A \cup \{ a_{new} \}$ *attr'*: $A' \rightarrow F \cup L \cup \{\perp\}$; *attr'*(a) := *attr*(a) $\forall$ a $\in$ A'\{ a_{new} }, *attr'*(a_{new}) := $\perp$ **Instances:** no effect, thus: $\mathfrak{I}'_{m'}$ = <R-UP, C, AV>

figure A-3: operation *insert attribute*

4. delete attribute

Criterion	Description
name of the operation	**delete_attribute**
informal explanation of semantics	deletes an existing isolated attribute. The attribute may not be connected to a dimension level or a fact.
syntax with input and output parameters	delete_attribute ($\mathcal{M}, \mathfrak{I}_m$, a_{del}) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, attribute a_{del} to be deleted **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$a_{del} \in A$, *attr*(a_{del}) = $\perp$.
post-condition(s)	$a_{del} \notin A$, $\mathfrak{I}'_{m'} = \mathfrak{I}_m$
example	delete_attribute ($\mathcal{M}, \mathfrak{I}_m$, "age")
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}'$ = <*F, L, A', gran, class, attr'*> A' = A - { a_{del} } *attr':* $A' \rightarrow F \cup L \cup \{\perp\}$; *attr'*(a) := *attr*(a) $\forall$ a $\in$ A' **Instances:** no effect, thus: $\mathfrak{I}'_{m'}$ = <R-UP, C, AV>

figure A-4: operation *delete attribute*

5. connect attribute to dimension level

Criterion	Description
name of the operation	**connect_attribute_to_dimension_level**
informal explanation of semantics	connects an existing attribute to an existing dimension level. A function *g* assigns values for the new attribute to every member of the dimension level.
syntax with input and output parameters	connect_attribute_to_dimension_level ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}$, a_{new}, l,g) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_{\mathcal{M}}$, attribute a_{new} to be connected, dimension level l to which a_{new} is connected, function g for the computation of the a_{new} values **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{\mathcal{M}'}$
pre-condition(s)	$a_{new} \in$A, $l \in$ L, attr(a_{new})=$\perp$, g must be well-defined for all dimension members of the level: g(m)=v with v$\in$ dom(a_{new}) $\forall$m $\in$ dom(l).
post-condition(s)	attr(a_{new})=l, av(a_{new}) is well-defined $\forall$m $\in$ dom(l)
example	connect_attribute_to_dimension_level ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}$, “age”, “customer”, “age(c)”)
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}'$=<F, L, A , *gran, class, attr'*> $attr' : A \rightarrow F \cup L \cup \{\perp\}$ $\quad attr'(a) := \begin{cases} l & if\ a = a_{new} \\ attr(a) & if\ a \neq a_{new} \end{cases}$ **Instances:** $\mathfrak{I}'_{\mathcal{M}'}$ = <R-UP, C, AV'>, AV': AV' := AV $\cup$ {av_{anew}}, define av_{anew}: dom (l) $\rightarrow$ dom(a_{new}) with $av_{a\,new}(m) := g(m)\ \forall m \in dom(l)$

figure A-5: operation *connect attribute to dimension level*

6. disconnect attribute from dimension level

Criterion	Description
name of the operation	**disconnect_attribute_from_dimension_level**
informal explanation of semantics	disconnects an attribute from a dimension level. Only the connecting edge is deleted, both the attribute and the dimension level still exist after execution of the operation.
syntax with input and output parameters	disconnect_attribute_from_dimension_level ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}$, a_{del}, l) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_{\mathcal{M}}$, attribute a_{del} to be disconnected, dimension level l to which a_{del} is yet connected **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{\mathcal{M}'}$
pre-condition(s)	$a_{del} \in A$, $l \in L$, $attr(a_{del})=l$
post-condition(s)	$a_{del} \in A$, $l \in L$, $attr(a_{del})=\perp$
example	disconnect_attribute_from_dimension_level ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}$, “age”, “customer”)
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}'$ =<F, L, A, *gran*, *class*, attr’> $attr' : A \rightarrow F \cup L \cup \{\perp\}$ $\quad attr'(a) := \begin{cases} \perp & \text{if } a = a_{del} \\ attr(a) & \text{if } a \neq a_{del} \end{cases}$ **Instances:** $\mathfrak{I}'_{\mathcal{M}'}$ = <R-UP, C, AV'>, AV': AV' := AV - { av_{adel} }with av_{adel} being the corresponding attribute value function for a_{del}

figure A-6: operation *disconnect attribute from dimension level*

7. connect attribute to fact

Criterion	Description
name of the operation	**connect_attribute_to_fact**
informal explanation of semantics	connects an existing attribute to an existing fact. A function g assigns values for the new attribute to every instance of the fact.
syntax with input and output parameters	connect_attribute_to_fact ($\mathcal{M}, \mathfrak{I}_m$, a_{new}, f, g) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, attribute a_{new} to be connected, fact f to which a_{new} is to be connected, function g for the computation of the a_{new} values **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$a_{new} \in$ A, $f \in$ F, attr(a_{new})=$\perp$, g must be well-defined for all fact instances
post-condition(s)	$a_{new} \in$ A, $f \in$ F, attr(a_{new})=f, c_f is well-defined for all fact instances
example	connect_attribute_to_fact ($\mathcal{M}, \mathfrak{I}_m$, “duration”, “vehicle repair”)
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}'$ =<F, L, A , *gran, class, attr'*> $attr' : A \rightarrow F \cup L \cup \{\perp\}$ $\quad attr'(a) := \begin{cases} f & if\ a = a_{new} \\ attr(a) & if\ a \neq a_{new} \end{cases}$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP, C', AV>, C' := C – {c_f} $\cup$ {c_f'} with c_f being the existing cube for f ; define c_f': dom(f) $\rightarrow$ codom(f) as $c'_f(x) := (z_1, \ldots, z_n, z_{n+1})$ *with* $(z_1, \ldots, z_n) = c_f(x)$ *and* $z_{n+1} = g(x)$

figure A-7: operation *connect attribute to fact*

8. disconnect attribute from fact

Criterion	Description
name of the operation	**disconnect_attribute_from_fact**
informal explanation of semantics	disconnects an attribute from a fact. Only the connecting edge is deleted, both the attribute and the fact still exist after execution of the operation.
syntax with input and output parameters	disconnect_attribute_from_fact ($\mathcal{M}, \mathfrak{I}_m$, a_{del}, f) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, attribute a_{del} to be disconnected, fact f to which a_{del} is yet connected **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$a_{del} \in$ A, $f \in$ F, $attr(a_{del})=f$
post-condition(s)	$a_{del} \in$ A, $f \in$ F, $attr(a_{del})=\perp$
example	disconnect_attribute_from_fact ($\mathcal{M}, \mathfrak{I}_m$, “duration”, “vehicle repair”)
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}'$ =<F, L, A, gran, class, attr’> $attr' : A \rightarrow F \cup L \cup \{\perp\}$ $\quad attr'(a) := \begin{cases} \perp & \text{if } a = a_{del} \\ attr(a) & \text{if } a \neq a_{del} \end{cases}$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP, C’, AV>, C’ := C – {c_f} ∪ {c_f'} with c_f being the existing cube for f ; define c_f': dom(f) → codom(f) as $c'_f(x) := (z_1, \ldots, z_{n-1})$ *with* $(z_1, \ldots, z_{n-1}, z_n) = c_f(x)$

figure A-8: operation *disconnect attribute from fact*

9. insert classification relationship

Criterion	Description
name of the operation	**insert_classification**
informal explanation of semantics	connects two existing dimension levels by a classification relationship. The dimension levels may both be isolated or already connected to other elements of the MD schema. If at least one of the dimension levels contains no instances yet, the corresponding classification relationship for the instances has to be defined.
syntax with input and output parameters	insert_classification ($\mathcal{M}, \mathfrak{I}_m$, l_1, l_2) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, two dimension level names l_1, l_2 to be connected. **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$l_1 \in$ L, $l_2 \in$ L, $\{(l_1,l_2)\} \notin$ class, $\{(l_2,l_1)\} \notin$ class. The classification relationship *class* between the instances must be well-defined
post-condition(s)	$\{(l_1,l_2)\} \in$ class, class is well-defined
example	insert_classification ($\mathcal{M}, \mathfrak{I}_m$, “month”, “year”)
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}' = <F, L, A, gran, class', attr>$ $class' = class \cup \{(l_1,l_2)\}$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP', C, AV>, R-UP' := R-UP $\cup$ { $r-up_{l1}^{l2}$ }, $\forall$ m $\in$ dom(l_1): $r-up_{l1}^{l2}$ (m):= k with k$\in$ dom(l_2). Additionally, $r-up_{l1}^{l2}$ (dom(l_1))$\subseteq$ dom(l_2), i.e., $r-up_{l1}^{l2}$ is well-defined $\forall$ m $\in$ dom(l_1).

figure A-9: operation *insert classification relationship*

10. delete classification relationship

Criterion	Description
name of the operation	**delete_classification**
informal explanation of semantics	disconnects an existing classification relationship between two dimension levels. The dimension levels are not deleted.
syntax with input and output parameters	delete_classification ($\mathcal{M}, \mathfrak{I}_m, l_1, l_2$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, two dimension level names l_1, l_2 to be disconnected. **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$l_1 \in$ L, $l_2 \in$ L, $\{(l_1,l_2)\} \in$ class
post-condition(s)	$\{(l_1,l_2)\} \notin$ class
example	delete_classification ($\mathcal{M}, \mathfrak{I}_m$, “month”, “year”)
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}'$ =<*F, L, A, gran, class’, attr*> *class*’ = *class* – $\{(l_1,l_2)\}$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP', C, AV>, R-UP' := R-UP - { $r - up_{l1}^{l2}$ }

figure A-10: operation *delete classification relationship*

11. insert fact

Criterion	Description
name of the operation	**insert_fact**
informal explanation of semantics	inserts a new, isolated fact.
syntax with input and output parameters	insert_fact ($\mathcal{M}, \mathfrak{I}_m, f$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, new fact name f **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$f_{new} \notin F$
post-condition(s)	$f_{new} \in F$
example	insert_fact ($\mathcal{M}, \mathfrak{I}_m$, "vehicle sales")
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}' = \langle F', L, A, gran', class, attr' \rangle$ $F' := F \cup \{f_{new}\}$, $gran': F' \rightarrow 2^L$, $gran'(f) := \begin{cases} \varnothing & if \quad f = f_{new} \\ gran(f) & if \quad f \neq f_{new} \end{cases}$ $attr' := attr$ **Instances:** $\mathfrak{I}'_{m'} = \langle$R-UP, C', AV$\rangle$, $C' := C \cup \{ c_{f\,new} \}$, define $c_{f\,new}$: dom (f_{new}) $\rightarrow$ codom(f_{new}) as c(x):= $\perp \forall$ x$\in$ dom(f_{new})

figure A-11: operation *insert fact*

12. delete fact

Criterion	Description
name of the operation	**delete_fact**
informal explanation of semantics	removes an existing, but isolated fact. Instances are deleted automatically.
syntax with input and output parameters	delete_fact ($\mathcal{M}, \mathfrak{I}_m$, f_{del}) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_m$, fact name f_{del} **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{m'}$
pre-condition(s)	$f_{del} \in F$, $gran(f_{del})=\emptyset$, $attr(a) \neq f_{del} \; \forall \; a \in A$
post-condition(s)	$f_{del} \notin F$, $c_{fdel} \notin C'$
example	delete_fact ($\mathcal{M}, \mathfrak{I}_m$, "vehicle sales")
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}' = <F', L, A, gran\|_{F'}, class, attr' >$ $F' := F - \{f_{del}\}$, $attr' := attr$ **Instances:** $\mathfrak{I}'_{m'}$ = <R-UP, C', AV>, $C' := C - \{ c_{f\,del} \}$

figure A-12: operation *delete fact*

13. insert dimension into fact

Criterion	Description
name of the operation	**insert_dimension_into_fact**
informal explanation of semantics	inserts an existing dimension (specified by a dimension level) into an existing fact, thus increasing the number of dimensions by one. A function *nv* has to be provided defining how the new values for the fact can be computed based upon the now extended set of dimensions and the old value of the fact. Each cell of the old cube now becomes a set of cells, exactly reflecting the new dimension.
syntax with input and output parameters	insert_dimension_into_fact ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}, l, f_{ins}, nv$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_{\mathcal{M}}$, level name *l* and fact name f_{ins} to be connected. Function *nv* to compute the distribution of existing fact instances over the new dimension. **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{\mathcal{M}'}$
pre-condition(s)	$l \in L$, $f_{ins} \in F$, $\{l\} \not\subset gran(f_{ins})$. The function *nv* must be well-defined for all existing fact instances
post-condition(s)	$l \in gran(f)$. The existing fact instances have been adapted w.r.t the new dimension according to function *nv*.
example	insert_dimension_into_fact ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}$, "customer", "vehicle sales", "sales_for_customer(c)")
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}' = < F, L, A, gran', class, attr >$ *define* $gran': F \rightarrow 2^L$ *as* $gran'(f) := \begin{cases} gran(f) & \text{for } f \neq f_{ins} \\ gran(f) \cup \{l\} & \text{for } f = f_{ins} \end{cases}$ **Instances:** $\mathfrak{I}'_{\mathcal{M}'}$ = <R-UP, C', AV> C' := C – $\{c_f\} \cup \{c_f'\}$ with c_f denoting the existing cube for f_{ins}. Although the fact f_{ins} itself does not change, its domain changes and the values of its co-domain have to be adapted. Consequently, we define a new cube c_f' and speak of f (or dom(f), codom(f)) if we refer to c_f and speak of f ' (or dom(f '), codom(f ')) if we refer to c_f'. We assume a dimensionality of *n* for c_f and a dimensionality of *n+1* for c_f'. c_f' is derived from c_f as follows: first, we compute the instances of dom(f '): for every combination $(x_1,...,x_n,x_{n+1}) \in$ dom(f) in c_f, add | dom(l) | new cube cells $(x_1,...,x_n,x_{n+1}, y)$ with $y \in$ dom(l) to c_f'.

	Second, compute the instances of codom(f '), i.e. adapt the measures: $c'_f(x_1,\ldots,x_n,x_{n+1}) = nv(c_f(x_1,\ldots,x_n),x_{n+1})$ with $nv : codom(f) \; x \; dom(l) \rightarrow codom(f')$ being the function that distributes the existing measures over the new dimension.

figure A-13: operation *insert dimension into fact*

14. delete dimension from fact

Criterion	Description
name of the operation	**delete_dimension**
informal explanation of semantics	deletes a dimension, specified by the dimension level, from a fact. The operation disconnects the base level *l* for this dimension from the fact f_{del}. Neither the fact nor the dimension level are deleted implicitly. Since the dimensionality of the fact is reduced, an aggregation function *agg* has to be provided which defines how the existing measures are aggregated over the deleted dimension (e.g. by summation).
syntax with input and output parameters	delete_dimension ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}, l, f_{del}, agg$) **input:** schema $\mathcal{M}$, instances $\mathfrak{I}_{\mathcal{M}}$, level name *l* and fact name f_{del} to be disconnected. Function *agg* to aggregate the existing fact instances over the deleted dimension **output:** new schema $\mathcal{M}'$, new instances $\mathfrak{I}'_{\mathcal{M}'}$
pre-condition(s)	$l \in L$, $f_{del} \in F$, $\{l\} \subseteq gran(f_{del})$. The function *agg* must be well-defined for all existing fact instances.
post-condition(s)	$l \in L$, $f_{del} \in F$, $\{l\} \not\subset gran(f_{del})$
example	delete_dimension ($\mathcal{M}, \mathfrak{I}_{\mathcal{M}}$, "customer", "vehicle sales")
semantics expressed by means of the MD data model	**Schema:** $\mathcal{M}' = < F, L, A, gran', class, attr >$ *define gran'*: $F \rightarrow 2^L$ *as* $gran'(f) := \begin{cases} gran(f) & for \quad f \neq f_{del} \\ gran(f) - \{l\} & \quad\quad f = f_{del} \end{cases}$ **Instances:** $\mathfrak{I}'_{\mathcal{M}'}$ = <R-UP, C', AV> C' := C – $\{c_f\} \cup \{c_f'\}$ with c_f denoting the existing cube for f_{del}. Although the fact f_{del} itself does not change, its domain changes and the values of its co-domain have to be adapted. Consequently, we again define a new cube c_f' and speak of f (or dom(f), codom(f)) if we refer to c_f and speak of f ' (or dom(f '), codom(f ')) if we refer to c_f'. We assume a dimensionality of *n* for c_f and a dimensionality of *n-1* for c_f'. We further assume that the dimension to be deleted corresponds to the *n*-th element in *dom(f)*. c_f' is derived from c_f as follows: c_f' *: dom (f ') → codom (f ') with dom(f ') being the reduced domain and* $c_f'(x_1, \ldots, x_{n-1}) = agg_{x_n}(c_f(x_1, \ldots, x_n)) \quad with \; x_n \in dom(l)$

figure A-14: operation *delete dimension*

Appendix B: Logical Evolution Operations

This appendix presents the logical evolution operations. As refinement to chapter 4.4.4, the description given here is the precise formal specification of the semantics of the logical evolution operations.

1. insert measure column$_L$

insert measure column$_L$	
Parameters: (see figure 3-12;figure 3-16)	measure m, fact f, instance adaptation function g
Composition:	insert attribute (m); connect attribute to fact (m,f,g)
Preconditions:	The fact table Ft_<f> exists and does not contain yet a column labeled m (assured by the conceptual preconditions: $f \in F$, $m \notin A$)
Schema transformation:	`ALTER TABLE Ft_<f> ADD COLUMN m`
Instance adaptation:	`UPDATE Ft_<f> SET m:= g(`d_1`, ..., `d_n`)` `WHERE Ft_<f>.dl`$_i$`=d`$_i$ with $dl_i \in FK_f$ $\forall i=1,...,n$[18]

Meta schema update:	`[INSERT INTO COLUMNS (cid, m, Ft_<f>)]`[19]
	`INSERT INTO MEASURES`
	`(m, f, dom(m), cid);`

figure B-1: operation *insert measure column$_L$*

2. delete measure column$_L$

delete measure column$_L$	
Parameters: (see figure 3-17;figure 3-13)	attribute m, fact f
Composition:	disconnect attribute from fact (m,f); delete attribute (m)
Preconditions:	The fact table Ft_<f> exists and contains a column labeled m (assured by the conceptual preconditions: $f \in F$, $m \in A$, attr(m) = f)
Schema transformation:	`ALTER TABLE Ft_<f> DROP COLUMN m`
Instance adaptation:	- (implicitly done by schema transformation)

Meta schema update:	`[DELETE FROM COLUMNS (cid, m, Ft_<f>)]`
	`DELETE FROM MEASURES`
	`(m, f, dom(m), cid);`

figure B-2: operation *delete measure column$_L$*

[18] This rather inefficient SQL code is better suited for defining the semantics of the instance adaptation. In a real implementation, we would suggest the use of temporary spool tables to generate more efficient SQL code.

[19] We remark that this update of the system catalogue is done by the DBMS when it processes the ADD COLUMN command of the schema transformation. In a real implementation, we would search the cid value in the system catalogue. Consequently, we set all these commands in brackets ("[ ]")to show that they are executed implicitly by the DBMS.

3. insert attribute column$_L$

insert attribute column$_L$	
Parameters: (see figure 3-12;figure 3-14)	attribute a, dimension level l, instance adaptation function g
Composition:	insert attribute (a); connect attribute to dim_level (a,l,g)
Preconditions:	At least one dimension table Dt_<bl> exists which contains a column for dimension level l. Each Dt_<bl> does not contain yet a column labeled a (assured by the conceptual preconditions: $l \in L$, $a \notin A$)
Schema transformation:	extend all dimension tables (denoted by Dt_<bl>) to which l belongs, as follows: `ALTER TABLE Dt_<bl> ADD COLUMN a`
Instance adaptation:	for all dimension tables (denoted by Dt_<bl>) to which l belongs: `UPDATE Dt_<bl> SET` a:= g(l) `UPDATE Dt_<bl> SET` a:= g(l_1,..., l_n,a_1,..,a_m) `WHERE Dt_<bl>.dl`$_i$`=l`$_i$ with $dl_i \in D_l$ $\forall i=1,...,n$ `AND Dt_<bl>.a`$_j$`=a`$_j$ with $a_j \in$ Attributes(D_l) $\forall j=1,...,m$

Meta schema update:	for all dimension tables (denoted by Dt_<bl>) to which l belongs:
	`[INSERT INTO COLUMNS`
	`(cid`$_i$`, a, Dt_<bl>);]`
	`INSERT INTO ATTRIBUTEMAPPING`
	`(a, cid`$_i$`);`
	additionally:
	`INSERT INTO ATTRIBUTES`
	`(a, l, dom(a) );`

figure B-3: operation *insert attribute column$_L$*

4. delete attribute column$_L$

delete attribute column$_L$	
Parameters: (see figure 3-13;figure 3-15)	attribute a, dimension level l
Composition:	disconnect attribute from dim_level (a,l); delete attribute (a)
Preconditions:	At least one dimension table Dt_<bl> exists which contains a column for dimension level l and a column for attribute a (assured by the conceptual preconditions: $l \in L$, $a \in A$, attr(a) = l)
Schema transformation:	for all dimension tables (denoted by Dt_<bl>) to which l belongs: `ALTER TABLE Dt_<bl> DROP COLUMN a`
Instance adaptation:	- (implicitly done by schema transformation)

Meta schema update:	`[DELETE FROM COLUMNS WHERE name = a;]`
	`DELETE FROM ATTRIBUTEMAPPING`
	`WHERE attribute = a;`
	`DELETE FROM ATTRIBUTES WHERE name = a;`

figure B-4: operation *delete attribute column$_L$*

5. insert fact table with dimension table$_L$

insert fact table with dimension table$_L$	
Parameters: (see figure 3-10; figure 3-20; figure 3-23)	fact f, dimension level l
Composition:	insert fact (f); insert level (l); insert dimension into fact (f, l)
Preconditions:	Neither a dimension table named Dt_<l> nor a fact table named Ft_<f> exist (assured by the conceptual preconditions: $l \notin L$, $f \notin F$)
Schema transformation:	`CREATE TABLE Dt_<l> (l:dom(l));` `CREATE TABLE Ft_<f> (l:dom(l));`
Instance adaptation:	Since both the dimension level and the fact are new, there are no *existing* instances to be adapted. We assume that new instances are inserted outside the scope of our schema design task.

Meta schema update:	`[INSERT INTO TABLES (Dt_<l>, Ft_<f>)]`
	`[INSERT INTO COLUMNS (cid`$_1$`,l,Dt_<l>),`
	`(cid`$_2$`,l, Ft_<f>)]`
	`INSERT INTO DIMENSION_LEVELS`
	`(l, TRUE, Dt_<l>, dom(l));`
	`INSERT INTO FACTS (f, Ft_<f>);`
	`INSERT INTO FACTHASDIM (f,l);`
	`INSERT INTO FACTDIMSMAPPING (l, cid`$_2$`);`
	`INSERT INTO DIMHIERARCHYMAPPING (l, cid`$_1$`)`

figure B-5: operation *insert fact table with dimension table$_L$*

6. insert fact table$_L$

insert fact table$_L$	
Parameters: (see figure 3-20;figure 3-23)	fact f, dimension level l, instance adaptation function nv
Composition:	insert fact (f); insert dimension into fact (f, l,nv)
Preconditions:	A dimension table exists that contains a column representing dimension level l. l may be the base level (i.e., the dimension table is labeled Dt_<l>) or not (i.e., the dimension table is labeled Dt_<bl>). No fact table labeled Ft_<f> exists. This is assured by the conceptual preconditions $l \in L$, $f \notin F$.

Schema transformation:	▪ Case 1: l is (or has been) base level of another fact:
	`CREATE TABLE Ft_<f> (l:dom(l));`
	▪ Case 2: l is not base level of any fact:
	`CREATE TABLE Dt_<l>`
	with attributes $D_l \cup$ Attributes(D_l), i.e. the attributes consist of l and all dimension levels above l in the classification hierarchy together with all describing attributes of these levels.
	`CREATE TABLE Ft_<f> (l:dom(l));`

Instance adaptation:	▪ Case 1: l is (or has been) base level of another fact: no adaptation of *existing* instances necessary.
	▪ Case 2: a new dimension table Dt_<l> has been created:
	Since l is not base level of any fact, l already belongs to another dimension table. Let us denote this dimension table as Dt_<x>. We further denote
	D_l:={l_1, ..., l_n }with l_1=l , $n \geq 1$ and
	Attributes(D_l):= {$a_{l.1}$, .., $a_{l.k}$, ... , $a_{ln.1}$, ..., $a_{ln.m}$}.
	First, we have to copy the distinct values for l:
	`INSERT INTO Dt_<l> (COLUMN l)`
	`SELECT DISTINCT l FROM Dt_<x>;`
	Then, we update the other copied dimension levels (if any) and all describing attributes:
	for each $l_i \in D_l$, i=**2**, ...,n (remember: l_1=l):
	`UPDATE Dt_<l> SET` $l_i := r - up^{l_i}_{l_{i-1}}(l_{i-1})$
	and for each $a_{li.j} \in$ Attributes(D_l), i=**1**, ...,n:
	`UPDATE Dt_<l> SET` $a_{li.j} := av_{a_{li.j}}(l_i)$

Meta schema update:	▪ Case 1: l is (or has been) base level of another fact: [INSERT INTO TABLES (Ft_<f>)] [INSERT INTO COLUMNS (cid_1,l, Ft_<f>)] INSERT INTO FACTS (f, Ft_<f>); INSERT INTO FACTHASDIM (f,l); INSERT INTO FACTDIMSMAPPING (l, cid_1); INSERT INTO DIMHIERARCHYMAPPING (l, cid_2) with cid_2 being the cid of l in Dt_<l>. ▪ Case 2: l is not base level of any fact: **additionally** to case 1: [INSERT INTO TABLES (Dt_<l>)] [INSERT INTO COLUMNS ((cid_j,m, Dt_<l>) $\forall$ m $\in D_l \cup$ Attributes(D_l), j = 2, ... , $\lvert D_l \cup \text{Attributes}(D_l)\rvert$-1] UPDATE DIMENSIONLEVELS SET is_base=TRUE WHERE name=l UPDATE DIMENSIONLEVELS SET table_name=Dt_<l> WHERE name=l

figure B-6: operation *insert fact table*$_L$

7. delete fact table$_L$

delete fact table$_L$	
Parameters: (see figure 3-24;figure 3-21)	fact f, dimension level l. The aggregation function agg is not used because the fact is deleted, too.
Composition:	delete dimension (l,f,agg); delete fact (f)
Preconditions:	A fact table labeled Ft_<f> with a single column l referencing the corresponding dimension table Dt_<l> exists (assured by the conceptual preconditions $l \in L$, $f \in F$, $l \in gran(f)$). The fact must not be connected to any other elements than the dimension level l. This precondition is guaranteed because a delete fact operation may only occur after the last edge is deleted from this fact (see remark concerning the ordering of schema evolution operations)
Schema transformation:	`DROP TABLE Ft_<f>;`
Instance adaptation:	- (implicitly done by schema transformation)

Meta schema update:	`[DELETE FROM TABLES WHERE name=Ft_<f>]`
	`[DELETE FROM COLUMNS WHERE table_name = Ft_<f>;]`
	`DELETE FROM FACTS WHERE name=f;`
	`DELETE FROM FACTHASDIM WHERE fact=f AND dim_level=l;`
	`DELETE FROM FACTDIMSMAPPING WHERE dim_Level=l and column_ID = cid;`
	with cid referencing the column in Ft_<f>
	(we remark that this operation should be executed before the schema transformation because otherwise cid would not be available any more)

figure B-7: operation *delete fact table$_L$*

8. insert dimension table$_L$

insert dimension table$_L$	
Parameters: (see figure 3-10;figure 3-23)	fact f, dimension level l
Composition:	insert level (l); insert dimension into fact (f, l)
Preconditions:	A fact table Ft_<f> exists and does not yet contain a column named l (assured by the conceptual preconditions: $l \notin L$, $f \in F$)

Schema transformation:	`CREATE TABLE Dt_<l> (l:dom(l));`
	`ALTER TABLE Ft_<f>`
	`ADD COLUMMN (l:dom(l));`

Instance adaptation:	Since the dimension level is new, there are no *existing* instances in the fact table or dimension table to be adapted. We assume that new instances are inserted outside the scope of our schema design task.

Meta schema update:	`[INSERT INTO TABLES (Dt_<l>;]`
	`[INSERT INTO COLUMNS (cid`$_1$`,l,Dt_<l>),`
	`(cid`$_2$`,l, Ft_<f>)]`
	`INSERT INTO DIMENSION_LEVELS`
	`(l, TRUE, Dt_<l>, dom(l));`
	`INSERT INTO FACTHASDIM (f,l);`
	`INSERT INTO FACTDIMSMAPPING (l, cid`$_2$`);`
	`INSERT INTO DIMHIERARCHYMAPPING (l, cid`$_1$`)`

figure B-8: operation *insert dimension table$_L$*

9. insert dimension level column$_L$

insert dimension level column$_L$	
Parameters: (see figure 3-10;figure 3-18)	dimension levels l_1, l_2, instance adaptation function $r-up_{l1}^{l2}$. Level l_1 is the level to which the new level l_2 is being connected.
Composition:	insert level (l_2); insert classification (l_1, l_2, $r-up_{l1}^{l2}$)
Preconditions:	At least one dimension table Dt_<bl> exists which contains a column representing l_1, but no column named l_2 (assured by : $l_1 \in L$, $l_2 \notin L$)

Schema transformation:	extend all dimension tables (denoted by Dt_<bl>) to which l_1 belongs, as follows:
	`ALTER TABLE Dt_<bl>`
	`ADD COLUMMN (`l_2`:dom(`l_2`));`

Instance adaptation:	for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: `UPDATE Dt_<bl> SET` $l_2 := r-up_{l1}^{l2}(l_1)$

Meta schema update:	for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs:
	`[INSERT INTO COLUMNS (cid`$_i$`,`l_2`,Dt_<bl>);]`
	`INSERT INTO DIMHIERARCHYMAPPING`
	`(`l_2`, cid`$_i$`)`
	additionally:
	`INSERT INTO DIMENSION_LEVELS`
	`(`l_2`, FALSE, NULL, dom(`l_2`));`
	`INSERT INTO CLASSIFICATIONS(`l_1`,`l_2`);`

figure B-9: operation *insert dimension level column$_L$*

10. insert classification$_L$

insert classification$_L$	
Parameters: (see figure 3-18)	dimension levels l_1, l_2 to be connected. The new classification relationship means that level l_1 can be classified according to level l_2.
Composition:	insert classification (l_1, l_2)
Preconditions:	There exists at least one dimension table which contains a column named l_1 . There exists also at least one dimension table (possibly the same) which contains a column named l_2 . This is assured by the conceptual preconditions: $l_1 \in L$, $l_2 \in L$.

Schema transformation:	extend all dimension tables (denoted by Dt_<bl>) to which l_1 belongs, as follows:
	▪ Case 1: l_2 exists already in Dt_<bl>:
	if l_2 is marked for deletion: unset deletion flag
	▪ Case 2: l_2 exists in another dimension table:
	`ALTER TABLE Dt_<bl>`
	`ADD COLUMMN (`c_{new}`:dom(`c_{new}`));`
	for all $c_{new} \in D_{l2} \cup$ Attributes(D_{l2}), i.e. the new columns consist of l_2 and all dimension levels above l_2 in the classification hierarchy together with all describing attributes of these levels.

Instance adaptation:	for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: ▪ Case 1: l_2 exists already in Dt_<bl>: no instance adaptation necessary. ▪ Case 2: l_2 exists in another dimension table: Let us denote D_{l2}:={l_2, ..., l_n }and Attributes(D_{l2}):= {$a_{l2.1}$, .., $a_{l2.k}$, ... , $a_{ln.1}$, ..., $a_{ln.m}$} for each $l_i \in D_{l2}$, i=2, ...,n: `UPDATE Dt_<bl> SET` $l_i := r - up^{l_i}_{l_{i-1}}(l_{i-1})$ and for each $a_{li.j} \in$ Attributes(D_{l2}), i=2, ...,n: `UPDATE Dt_<bl> SET` $a_{li.j} := av_{a_{li.j}}(l_i)$

Meta schema update:	for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs:
	■ Case 1: l_2 exists already in Dt_<bl>:
	`INSERT INTO CLASSIFICATIONS(`l_1`,`l_2`);`
	■ Case 2: l_2 exists in another dimension table:
	for each $l_i \in D_{l2}$, i=2, ...,n:
	`[INSERT INTO COLUMNS`
	`(cid`$_j$`,`l_i`,Dt_<bl>);]`
	`INSERT INTO DIMHIERARCHYMAPPING`
	`(`l_i`, cid`$_j$`)`
	and for each $a_{li.j} \in$ Attributes(D_{l2}), i=2, ...,n:
	`[INSERT INTO COLUMNS`
	`(cid`$_k$`,` $a_{li.j}$ `,Dt_<bl>);]`
	`INSERT INTO ATTRIBUTEMAPPING`
	`(`$a_{li.j}$`, cid`$_k$`)`
	`INSERT INTO CLASSIFICATIONS(`l_1`,`l_2`);`

figure B-10: operation *insert classification*$_L$

11. delete classification$_L$

delete classification$_L$	
Parameters: (see figure 3-19)	dimension levels l_1, l_2 to be disconnected.
Composition:	delete classification (l_1, l_2)
Preconditions:	There exists at least one dimension table which contains a column labeled l_1 and at least one dimension table which contains a column labeled l_2 (assured by the conceptual pre-conditions $l_1 \in L$, $l_2 \in L$, (l_1, l_2) $\in$ class).
Schema transformation:	for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs: mark all dimension levels $\in D_{l2}$ and all attributes $\in$ Attributes(D_{l2}) for deletion.
Instance adaptation:	No instance adaptation when processing this operation. The instances of l_2 are either needed again when this level is connected elsewhere, deleted when l_2 is deleted, or deleted from the dimension table(s) Dt_<bl> during the garbage collection at the end of the processing phase in the transformation algorithm.

Meta schema update:	`DELETE FROM CLASSIFICATIONS`
	`WHERE dim_level1=`l_1 `AND dim_level2=`l_2`;`
	for all dimension tables (denoted by Dt_<bl>) to which l_1 belongs:
	for each $l_i \in D_{l2}$, i=2, ...,n:
	mark all tuples in `COLUMNS WHERE name=` l_i for deletion
	mark all tuples in `DIMHIERARCHYMAPPING`
	`WHERE dim_level =` l_i for deletion
	for each $a_{li.j} \in$ Attributes(D_{l2}), i=2, ...,n:
	mark all tuples in `COLUMNS WHERE name=`$a_{li.j}$ for deletion
	mark all tuples in `ATTRIBUTEMAPPING`
	`WHERE attribute =` $a_{li.j}$ for deletion

figure B-11: operation *delete classification$_L$*

12. delete dimension level column$_L$

delete dimension level column$_L$	
Parameters: (see figure 3-11)	dimension level l
Composition:	delete level (l)
Preconditions:	At least one dimension table exists which contains a column labeled l. l may be the base level of this dimension table (i.e. the dimension table is named Dt_<l>). This is assured by the conceptual precondition $l \in L$.

Schema transformation:	for all dimension tables (denoted by Dt_<bl>) to which l belongs:
	`if (l is the only level of this dimension table)`
	`  DROP TABLE Dt_<bl>`
	`else`
	`  ALTER TABLE Dt_<bl> DROP COLUMN l`

Instance adaptation:	for all dimension tables (denoted by Dt_<bl>) to which l belongs:
	If l = bl (i.e. l is the base level of this dimension table) and
	l is not the only level of Dt_<bl>:
	/* Duplicates in the new base level have to be eliminated:
	/* let Dt_<bl> $\subseteq$ bl $\times$ l_1 $\times$ ... $\times$ l_k $\times$ a_1 $\times$ ... $\times$ a_m
	`INSERT INTO TEMP`
	`SELECT DISTINCT` l_1, ..., l_k, a_1, ... , a_m
	`FROM Dt_<bl>;`
	`DELETE FROM Dt_<bl>;`
	`INSERT INTO Dt_<bl> SELECT * FROM TEMP;`
	
	All other cases are done implicitly by the schema transformation.

Meta schema update:	for all dimension tables (denoted by Dt_<bl>) to which l belongs:
	
	▪ Case 1: l is the only level of this dimension table:
	`[DELETE FROM COLUMNS WHERE name=l;]`
	`[DELETE FROM TABLES`
	`WHERE name= Dt_<bl>; ]`
	`DELETE FROM FACTDIMSMAPPING`
	`WHERE dim_level= l;`
	
	▪ Case 2: l is base level, but not the only level of this dimension table:
	`[DELETE FROM COLUMNS WHERE name=l;]`
	`DELETE FROM FACTDIMSMAPPING`
	`WHERE dim_level= l;`
	
	▪ Case 2: else:
	`[DELETE FROM COLUMNS WHERE name=l;]`
	
	Additionally in both cases:
	`DELETE FROM DIMHIERARCHYMAPPING`
	`WHERE dim_level=l;`
	`DELETE FROM DIMENSIONLEVELS`
	`WHERE name=l;`

figure B-12: operation *delete dimension level column*$_L$

13. insert dimension$_L$

insert dimension$_L$	
Parameters: (see figure 3-23)	fact f, dimension level l, instance adaptation function nv
Composition:	insert dimension into fact (f, l, nv)
Preconditions:	There exists at least one dimension table containing a column labeled l. The dimension table may be labeled Dt_<l> (if l is the base level) or Dt_<bl> (if l is not the base level). There exists a fact table Ft_<f> which does not yet contain a column labeled l (assured by the conceptual preconditions: $l \in L$, $f \in F$)

Schema transformation:	▪ Case 1: l is (or has been) base level of another fact:
	`MODIFY TABLE Ft_<f>`
	`ADD COLUMN (l:dom(l));`
	▪ Case 2: l is not base level of any fact:
	`CREATE TABLE Dt_<l>`
	with attributes $D_l \cup$ Attributes(D_l), i.e. the attributes consist of l and all dimension levels above l in the classification hierarchy together with all describing attributes of these levels.
	`MODIFY TABLE Ft_<f>`
	`ADD COLUMN (l:dom(l));`

Instance adaptation:	▪ only in case 2: a new dimension table Dt_<l> has been created:
	Since l is not base level of any fact, l already belongs to another dimension table. Let us denote this dimension table as Dt_<x>. We further denote
	D_l:={l_1, ..., l_n }with l_1=l , $n \geq 1$ and
	Attributes(D_l):= {$a_{l.1}$, .., $a_{l.k}$, ... , $a_{ln.1}$, ..., $a_{ln.m}$}.
	First, we have to copy the distinct values for l:
	`INSERT INTO Dt_<l> (COLUMN l)`
	`SELECT DISTINCT l FROM Dt_<x>;`
	Then, we update the other copied dimension levels (if any) and all describing attributes:
	for each $l_i \in D_l$, i=**2**, ...,n (remember: l_1=l):
	`UPDATE Dt_<l> SET` $l_i := r - up^{l_i}_{l_{i-1}}(l_{i-1})$
	and for each $a_{li.j} \in$ Attributes(D_l), i=**1**, ...,n:

	UPDATE Dt_<l> SET $a_{li.j}$:= $av_{a_{li.j}}$ (l_j) ■ in both cases, adapt the data in the fact table according to the increased dimensionality: Let us assume FK_f := {dl_1, ..., dl_n} and $Measure_f$:= {m_1, ..., m_k}. CREATE TABLE TEMP (dl_1, ..., dl_n, l, m_1, ..., m_k); INSERT INTO TEMP (SELECT F.dl_1, ..., F.dl_n, D.l, F.m_1, ... , F.m_k FROM Ft_<f> AS F, Dt_<l> AS D); UPDATE TEMP SET (m_1, ... , m_k):= nv(m_1, ... , m_k,l); DELETE FROM Ft_<f>; INSERT INTO Ft_<f> (SELECT * FROM TEMP WHERE (m_1, ... , m_k) NOT NULL);
Meta schema update:	■ Case 1: l is (or has been) base level of another fact: [INSERT INTO COLUMNS (cid_1,l, Ft_<f>)] INSERT INTO FACTHASDIM (f,l); INSERT INTO FACTDIMSMAPPING (l, cid_1); INSERT INTO DIMHIERARCHYMAPPING (l, cid_2) with cid_2 being the cid of l in Dt_<l>. ■ Case 2: l is not base level of any fact: **additionally** to case 1: [INSERT INTO TABLES (Dt_<l>)] [INSERT INTO COLUMNS ((cid_j,m, Dt_<l>) ∀ m ∈ D_l ∪ Attributes(D_l), j = 2, ... , | D_l ∪ Attributes(D_l)|-1] UPDATE DIMENSIONLEVELS SET is_base=TRUE WHERE name=l UPDATE DIMENSIONLEVELS SET table_name=Dt_<l> WHERE name=l

figure B-13: operation *insert dimension*$_L$

14. delete dimension$_L$

delete dimension$_L$	
Parameters: (see figure 3-24)	fact f, dimension level l, instance adaptation function agg
Composition:	delete dimension level from fact (l, f,agg)
Preconditions:	There exists a dimension table named Dt_<l> containing a column named l. There exists a fact table Ft_<f> with a column l referencing Dt_<l>. This is assured by the conceptual preconditions: $l \in L$, $f \in F$, $l \in gran(f)$.
Schema transformation:	We remark that the instance adpatation must be executed before the schema transformation takes place. Otherwise, necessary instance information would be lost. `MODIFY TABLE Ft_<f> DROP COLUMN l;`

Instance adaptation:	The data in the fact table has to be adapted according to the decreased dimensionality, i.e. the measure values have to be aggregated using the instance adaptation function agg:
	Let us assume $FK_f := \{dl_1, ..., dl_n\}$ and – without loss of generality $l = dl_n$. Further, we assume
	$Measure_f := \{m_1, ..., m_k\}$.
	We aggregate the data using a temporary table:
	`INSERT INTO TEMP`
	`  (SELECT F.dl`$_1$`, ..., F.dl`$_{n-1}$`,`
	`  agg (F.m`$_1$`, ... , F.m`$_k$`)`
	`  FROM Ft_<f> AS F`
	`  GROUP BY F.dl`$_1$`, ..., F.dl`$_{n-1}$`);`
	`DELETE FROM Ft_<f>;`
	`INSERT INTO Ft_<f> (SELECT * FROM TEMP);`

Meta schema update:	`[DELETE FROM COLUMNS WHERE name=l AND table_name=Ft_<f>)]`
	`DELETE FROM FACTHASDIM WHERE fact=f AND dim_level=l;`
	`DELETE FROM FACTDIMSMAPPING WHERE dim_level=l AND column_ID=cid;`
	with cid being the identifier of column l in Ft_<f>. We remark that this operation must be executed before the schema transformation takes place in order to avoid loss of necessary information.

figure B-14: operation *delete dimension$_L$*

References

[AGS97] R. Agrawal, A. Gupta, S. Sarawagi: *Modelling Multidimensional Databases.* Proceedings of the Thirteenth International Conference on Data Engineering (ICDE), Birmingham U.K., April 1997.

[ASU88] A. Aho, R. Sethi, J.D. Ullman: *Compilers. Principles, Techniques and Tools.* Addison Wesley, 1988.

[Bay96] R. Bayer: *The universal B-Tree for multidimensional indexing.* Technical Report TUM-I9637, Institut für Informatik, Technische Universität München, 1996.

[Bay97] R. Bayer: *The universal B-Tree for multidimensional indexing: General Concepts.* Proceedings of the International Conference on World-Wide Computing and Its Applications ’97 (WWCA 97), Tsukuba, Japan, Springer Lecture Notes on Computer Science, March 1997.

[Bel98] Z. Bellahsene: *View Adaptation in Data Warehousing Systems.* Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA), Vienna, Austria, August 1998.

[BeLi91] P.L. Bergstein, K.J. Liebherr: *Incremental Class Dictionary Learning and Optimization*. Proceedings European Conference on Object-Oriented Programming (ECOOP ’91), Geneva, Switzerland, July 1991.

[BG+00] A. Bauer, H. Günzel (eds.): *Data Warehouse - Architektur, Entwicklung, Anwendung*. dpunkt, Heidelberg, Germany, 2000.

[BKK+87] J. Banerjee, W. Kim, H.-J. Kim, H.F. Korth: *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*, Proceedings of the ACM SIGMOD Conference on Managament of Data, San Francisco, California, May 1987.

[BD94] P. A. Bernstein, U. Dayal: *An Overview of Repository Technology*. Proceedings of the 20th International Conference on Very Large Databases (VLDB), Santiago, Chile, 1994.

[BL97] A, Bauer, W. Lehner: *The Cube-Query-Language (CQL) for Multidimensional Statistical and Scientific Database Systems*. Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA), Melbourne, Australia, April 1997.

[Bla99] M. Blaschka: *FIESTA: A Framework for Schema Evolution in Multidimensional Information Systems*. Proceedings of the 6th CAiSE Doctoral Consortium, Heidelberg, Germany, June 1999.

[BSHD98] M. Blaschka, C. Sapia, G. Höfling, B. Dinter: *Finding your way through multidimensional data models*. Proceedings of the International Workshop on Data Warehouse Design and OLAP Technology (DWDOT, in connection with DEXA), Vienna, Austria, August 1998.

[BSH99] M. Blaschka, C. Sapia, G. Höfling: *On Schema Evolution in Multidimensional Databases*. Proceedings of First International Conference on Data Warehousing and Knowledge Discovery (DaWak'99), Florence, Italy, August 30 - September 1, 1999. Lecture Notes in Computer Science, Vol. 1676, Springer, 1999.

[BSH00] M. Blaschka, C. Sapia, G. Höfling: *BabelFish Project Report*. Internal Report, FORWISS, Munich, Germany, January 2000.

[Cas92] E. Casais: *An Incremental Class Reorganization Approach*. Proceedings of the European Conference on Object-Oriented Programming (ECOOP '92), Utrecht, The Netherlands, June 29 - July 3, 1992.

[CD97] S. Chaudhuri, U. Dayal: *An Overview of Data Warehousing and OLAP Technology*. SIGMOD Record 26(1), pp. 65-74, 1997

[Che76] P.P-S. Chen: *The Entity Relationship Model – Towards a Unified View of Data*. ACM Transaction on Database Systems (TODS) Vol. 1, No. 1, 1976

[CNR99] K. T. Claypool, C. Natarajan, E. A. Rundensteiner: *Optimizing the Performance of Schema Evolution Sequences*. Technical Report WPI-CS-TR-99-06, Worcester Polytechnic Institute, March 1999.

[Cog98a] Cognos Corporation: *Cognos Powerplay – Transformer MDL Reference*, Online Documentation, 1998

[Cog98b] Cognos Corporation: *Cognos Powerplay – Discovering Transformer*, Online Documentation, 1998

[Cog98c] Cognos Corporation: *Cognos Powerplay – Step By Step Transformer*, Online Documentation, 1998

[CS98] P. Chamoni, S. Stock: *Modellierung temporaler multidimensionaler Daten in Analytischen Informationssystemen*. In: Kruse, Saake (eds.): Data Mining und Data Warehousing, Arbeitsbericht 14, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, Magdeburg 1998.

[CS99] P. Chamoni, S. Stock: *Temporal Structures in Data Warehousing*. Proceedings of First International Conference on Data Warehousing and Knowledge Discovery (DaWaK '99), Florence, Italy, August 30 - September 1, 1999. Lecture Notes in Computer Science, Vol. 1676, Springer, 1999.

[CT97] L. Cabibbo, R. Torlone: *Querying Multidimensional Databases.* Proceedings of the 6th International Workshop on Database Programming Languages (DBPL), Estes Park, Colorado, USA, August 1997.

[CT98] L. Cabibbo, R. Torlone: *A Logical Approach to Multidimensional Databases.* Proceedings of the 6th International Conference on Extending Database Technology (EDBT), Valencia, Spain, March 1998.

[Dev97] B. Devlin: *Data Warehouse: from Architecture to Implementation.* Addison-Wesley, Reading, MA, USA, 1997.

[DKPW99] S. Dekeyser, B. Kuijpers, J. Paredaens, J. Wijsen: *The nested datacube model for OLAP.* In: Advances in Database Technology (Proceedings of the International Workshop on Data Warehousing and Data Mining – DWDM 98 in conjunction with the ER98 Conference on Coneptual Modeling), LNCS Vol 1552, Springer, Berlin, 1999.

[DSBH98] B. Dinter, C. Sapia, G. Höfling, M. Blaschka: *The OLAP Market: State of the Art and Research Issues*, Proceedings of First International Workshop on Data Warehousing and OLAP (DOLAP, in connection with CIKM'98), Washington, D.C., USA, November 1998.

[DSBH99] B. Dinter, C. Sapia, G. Höfling, M. Blaschka: *OLAP Market and Research: Initiating the Cooperation*, Journal of Computer Science and Information Management, Vol. 2, No. 3, 1999.

[DSVH97] B. Dinter, C. Sapia, M. Vrca, G. Höfling: *The OLAP Market: Architectures, Products, Trends* (in German). FORWISS Technical Report, Munich, September 1997.

[DT97] A. Datta ,H. Thomas: *A Conceptual Model and an algebra for On-Line Analytical Processing in Data Warehouses*, Proceedings of the 7th Workshop on Information, Technologies and Systems (WITS), Atlanta, Georgia, USA, December 1997.

[DZR99] L. Ding, X. Zhang, E.A. Rundensteiner: *Enhancing Existing Incremental View Maintenance Algorithms Using the Multi-Relation Encapsulation Wrapper*. Technical Report WPI-CS-TR-99-23, Worcester Polytechnic Institute, August 1999.

[Ehr79] H. Ehrig: *Introduction to the algebraic theory of graph grammars (a survey)*. Proceedings of the International Workshop on Graph Grammars and their Application to Computer Science and Biology, LNCS 73, Springer Verlag, 1979.

[Eic91] C.F. Eick: *A Methodology for the Design and Transformation of Conceptual Schemas*. Proceedings of the 17th International Conference on Very Large Databases, Barcelona, Spain, September 1991.

[GBL+96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh: *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Total.* Proceedings of the 12th International Conference on Data Engineering, New Orleans, Louisiana, USA, February/March 1996.

[GCB+97] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao: *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Total.* Data Mining and Knowledge Discovery, 1 (1):29-53, January 1997.

[GJM96] A. Gupta, H. V. Jagadish, I. S. Mumick: *Data Integration using Self-Maintainable Views*. Proceedings of the 5th International Conference on Extending Database Technology (EDBT), Avignon, France, March 1996.

[GL97] M. Gyssens, L.V.S. Lakshmanan: *A Foundation for Multi-Dimensional Databases*. Proceedings of 23rd International Conference on Very Large Data Bases, August 1997, Athens, Greece.

[GM95] A. Gupta, I. S. Mumick: *Maintenance of Materialized Views: Problems, Techniques, and Applications*, Data Engineering Bulletin, Vol. 18 No.2 , June 1995.

[GMR95] A. Gupta, I.S. Mumick, K.A. Ross: *Adapting Materialized Views after Redefinition*. Proceedings ACM SIGMOD International Conference on Management of Data, San José, California, USA, 1995.

[GMR98] M. Golfarelli, D. Maio, S. Rizzi: *Conceptual design of data warehouses from E/R schemes*, Proceedings 31st Hawaii International Conference on System Sciences (HICCS), Hawaii, USA, December 1998.

[GR98] M. Golfarelli, S. Rizzi, *A Methodological Framework for Data Warehouse Design*, Proceedings of First International Workshop on Data Warehousing and OLAP (DOLAP, in connection with CIKM'98), Washington, D.C., USA, November 1998.

[Gün00] H. Günzel: *Versioning for Data Warehouses – the TEMPS approach*. Proceedings of the workshop on temporal aspects, GI working group "Concepts of Data Warehousing", Universität Erlangen-Nürnberg, May 2000.

[Haa99] S. Haas: *Konzeption und Implementierung eines konfigurierbaren graphischen Modellierungswerkzeugs.* Master Thesis, FORWISS / Technische Universität München, Munich, Germany, August 1999.

[Hah00] K. Hahn: *Generierung von multidimensionalen Schemata aus konzeptuellen Modellen*. Master Thesis, FORWISS / Technische Universität München, Munich, Germany, February 2000.

[Har99] A. Harren: *Konzeptionelles Data Warehouse-Design*. Master Thesis, Universität Oldenburg, Oldenburg, Germany, May 1999.

[HBD+97] G. Höfling, M. Blaschka, B. Dinter, P. Spiegel, T. Ringel: *Data Warehouse Technology for the Management of Diagnosis Data* (in German). In Dittrich, Geppert (eds.): Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), Ulm, Germany, Springer Verlag, 1997.

[Her00] O. Herden: *A Design Methodology for data warehouses*. Proceedings of the 7th CAiSE Doctoral Consortium, Stockholm, Sweden, June 2000.

[HH99] A. Harren, O. Herden: *Conceptual Modeling of Data Warehouses*. Proceedings of the 18th International Conference on Conpetual Modeling (ER '99), Paris, France, November 1999.

[HMV99a] C.A. Hurtado, A.O. Mendelzon, A.A. Vaisman: *Maintaining Data Cubes under Dimension Updates.* Proceedings of the 15th International Conference on Data Engineering (ICDE), Sydney, Australia, March 1999.

[HMV99b] C.A. Hurtado, A.O. Mendelzon, A.A. Vaisman: *Updating OLAP Dimensions.* Proceedings of the 2nd International Workshop on Data Warehousing and OLAP, Kansas City, Missouri, USA, November 1999.

[Höf96] G. Höfling: *Schema-Evolution in objekt-orientierten Datenbanken*, Dissertation, Technische Universität München, Munich, Germany, 1996.

[HRU96] V. Harinarayan, A. Rajaraman, J. D. Ullman: *Implementing Data Cubes effi-*

ciently. Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada, June 1996.

[Inf98a] Informix Corporation: *Informix Dynamic Server, Version 7.3: Informix Guide to SQL*, Informix Press, Menlo Park, USA, February 1998.

[Inf98b] Informix Corporation: *Administrator's Guide for Informix Dynamic Server, Version 7.3*, Informix Press, Menlo Park, USA, February 1998.

[Inf98c] Informix Corporation: *Explorer User's Guide. MetaCube ROLAP Option for Informix Dynamic Server, Version 4.0*, Informix Press, Menlo Park, USA, January 1998.

[Inf98d] Informix Corporation: *Data Warehouse Administrator's Guide. MetaCube ROLAP Option for Informix Dynamic Server, Version 4.0*, Informix Press, Menlo Park, USA, January 1998.

[Inm96] W.H. Inmon: *Building the Data Warehouse*, 2nd Edition, Wiley, New York, USA. 1996.

[ISO82] International Standards Organization, ISO/TC97/SC5/WG3: *Concepts and Terminology for the Conceptual Schema and the Information Base*. Publication Number ISO/TC97/SC5/N695, 1982.

[ISO90] International Standards Organization: *IRDS Framework* ISO/IEC IS 10027, 1990.

[JGJ+95] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, S. Eherer: *ConceptBase – a deductive object base for meta data management.* Journal of Intelligent Information Systems, 4 (2), 1995.

[JJQ+99] M. Jarke, M.A. Jeusfeld, C. Quix, P. Vassiliadis: *Architecture and Quality in Data Warehouses: An Extended Repository Approach*. Information Systems, 24 (3), pp. 229-253, 1999.

[JLV+00] M. Jarke, M, Lenzerini, Y. Vassiliou, P. Vassiliadis (eds.): *Fundamentals of Data Warehouses*. Springer, Berlin, 2000.

[KhAb90] S. Khoshafian, R. Abnous: *Object Orientation: Concepts, Languages, Databases*, User Interfaces. Wiley, New York, USA, 1990.

[Kim96a] R. Kimball: *The Data Warehouse Toolkit*. Wiley, New York, 1996.

[Kim96b] R. Kimball: *Slowly Changing Dimensions*, Data Warehouse Architect, DBMS Magazine, April 1996, URL: http://www.dbmsmag.com

[KRH98] A. Koeller, E. A. Rundensteiner, N. Hachem: *Integrating the Rewriting and Ranking Phases of View Synchronization.* Technical Report WPI-CS-TR-98-23, Worcester Polytechnic Institute, December 1998.

[Kur99] A. Kurz: *Data Warehousing. Enabling Technology.* mitp Publishing, Bonn, Germany, 1999.

[Leh98] W. Lehner: *Modeling Large Scale OLAP Scenarios*. Proceedings of the 6th International Conference on Extending Database Technology (EDBT), Valencia, Spain, March 1998.

[LKN+98] A. Lee, A. Koeller, A. Nica and E. A. Rundensteiner: *Data Warehouse Evolution: Trade-offs between Quality and Cost of Query Rewritings.* Technical Report WPI-CS-TR-98-2, Worcester Polytechnic Institute, January 1998.

[LKN+99] A. Lee, A. Koeller, A. Nica and E. A. Rundensteiner: *Non-Equivalent Query Rewritings*, Proceedings of International Database Conference (IDC'99), Hong Kong, July, 1999.

[LRT96] W. Lehner, T. Ruf, M. Teschke: *CROSS-DB: A Feature-Extended Multidimensional Data Model for Statistical and Scientific Databases.* Proceedings of the 5th International Conference on Information and Knowledge Management (CIKM), Rockville, Maryland, USA, November 1996.

[LST99] J. Lewerenz, K.-D. Schewe, B. Thalheim: *Modelling Data Warehouses and OLAP Applications by Means of Dialogue Objects.* Proceedings of the Conference on Entity/Relationship Modeling (E/R '99), Paris, France, November 1999.

[LW96] C. Li, X.S. Wang: *A data model for supporting on-line analytical processing.* Proceedings of the 5th International Conference on Information and Knowledge Management (CIKM), Rockville, Maryland, USA, November 1996.

[Mar93] J. Marche: *Measuring the Stability of Data Models.* European Journal of Information Systems, Vol. 2, No. 1, 1993.

[Mar99] V. Markl: *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*. Dissertation, Technische Universität München, Munich, Germany, 1999.

[MCA+00] A. de Miguel, J. M. Cavero, J. Canela, A. S. de Miguel: *IDEA-DWCASE: Modeling Multidimensional Databases*. Proceedings of the Seventh International Conference on Extending Database Technology (EDBT), Konstanz, Germany, March 2000.

[McG96] F. McGuff: *Data Modelling for Data Warehouses*.
URL: http://members.aol.com/fmcguff/dwmodel/

[MD96] M. Mohania, G. Dong: *Algorithms for Adapting Materialized Views in Data Warehouses*, International Symposium on Cooperative Database Systems for Advanced Applications, Kyoto, Japan, World-Scientific, 1996.

[Moh97] M. Mohania: *Avoiding Re-computation: View Adaptation in Data Warehouses*. Proceedings 8th International Database Workshop, Hong-Kong, Springer LNCS, 1997

[MQM97] I. Mumick, D. Quass, M. Mumick: *Maintenance of data cubes and summary tables in a warehouse*. Proceedings ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, 1997.

[MRB99] V. Markl, F. Ramsak, R. Bayer: *Improving OLAP Performance by Multidimensional Hierarchical Clustering*. Proceedings of the International Database Engineering and Applications Symposium (IDEAS 1999), Montreal, Canada, August 1999.

[Nic99] A. Nica: *View Evolution Support for Information Integration Systems over Dynamic Distributed Information Spaces*. PhD thesis, University of Michigan, USA, 1999.

[NR99] A. Nica, E. A. Rundensteiner: *View Maintenance after View Synchronization*. Proceedings of the International Database Engineering and Application Symposium (IDEAS'99), Montreal, Canada, April 1999.

[Ora97] Oracle: *Express Administrator Guide*, Version 6.1, Oracle Corporation, 1997.

[PJ99] T. B. Pedersen, C. S. Jensen: *Multidimensional Data Modeling for Complex Data*. Proceedings of the 15th International Conference on Data Engineering (ICDE), Sydney, Austrialia, March 1999.

[QGM+96] D. Quass, A. Gupta, I. S. Mumick, J. Widom: *Making Views Self-Maintainable for Data Warehousing*. Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS), December 1996, Miami Beach, Florida, USA, 1996

[Qui99] C. Quix: *Repository Support for Data Warehouse Evolution*. Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW '99) in conjunction with CAiSE, Heidelberg, Germany, June 1999.

[Rati97] Rational Software Corp. & UML Partners: *UML Notation Guide, Version 1.1*. Object Management Group, OMG Document ad/97-08-05, 1997.

[Rati98] Rational Software Corp.: *Rational Rose 98*, URL: http://www.rational.com./rose, 1998.

[RKZ+99] E. A. Rundensteiner, A. Koeller, X. Zhang, A.J. Lee, A. Nica: *Evolvable View Environment EVE: A Data Warehouse System Handling Schema and Data Changes of Distributed Sources*. Proceedings of the International Database Engineering and Application Symposium (IDEAS'99), Montreal, Canada, April 1999.

[RLN97] E.A. Rundensteiner, A.J. Lee, A. Nica: *On Preserving Views in Evolving Environments.* Proceedings of the 4th Workshop on Knowledge Represenation Meets Databases (KRDB), Athens, Greece, August 1997.

[RS97] J. Rekers, A. Schürr: *Defining and Parsing Visual Languages with Layered Graph Grammars*. Journal of Visual Languages and Computing 8 (1): 27-55, 1997.

[Sap99] C. Sapia: *On Modeling and Predicting User Behavior in OLAP Systems*. Proceedings of the CaiSE99 Workshop on Design and Management of Data Warehouses 99 (DMDW99), Heidelberg, Germany, June 1999.

[Sap00] C. Sapia: *PROMISE-Modeling and Predicting User Behavior for Online Analytical Processing Applications*. Dissertation Draft, FORWISS, Munich, January 2000.

[SBHD98] C. Sapia, M. Blaschka, G. Höfling, B. Dinter, *Extending the E/R Model for the Multidimensional Paradigm*, in Advances in Database Technologies, Springer LNCS Vol 1552, Proceedings of the International Workshop on Data Warehousing and Data Mining – DWDM 98 in conjunction with the ER98 Conference on Conceptual Modeling, Singapore, November 1998.

[SBH99] C. Sapia, M. Blaschka, G. Höfling, *An Overview of Multidimensional Data Models*, FORWISS Technical Report FR-1999-001, http://www.forwiss.tu-muenchen.de/~system42/publications , Munich, Germany, January 1999.

[SBH00] C. Sapia, M. Blaschka, G. Höfling: *GraMMi: The Design and Implementation of a Generic Metadata-driven Graphical Modeling Tool*. Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS-33), 4-7 January 2000, Maui, Hawaii, USA.

[Sch93] B. Schiefer: *Eine Umgebung zur Unterstützung von Schemaänderungen und Sichten in objektorientierten Datenbanksystemen.* Dissertation, Universität Karlsruhe, 1993.

[Sim95] A.R. Simon: *Strategic Database Technology: Management for the Year 2000*. Morgan Kaufmann, San Francisco, CA, USA 1995.

[Sir97] H. Sirtl: *Modellierung und Aufbau einer Warehouse-Datenbank*. Diplomarbeit, FORWISS / Technische Universität München, Munich, Germany, May 1997.

[Sjo93] D. Sjoberg: *Quantifying Schema Evolution.* Information and Software Technology Journal, Vol. 35, No. 1, January 1993.

[SMK+98] S. Samtani, M. K. Mohania, V. Kumar, Y. Kambayashi: *Recent Advances and Research Problems in Data Warehousing*. In: Yahiko Kambayashi, Dik Lun Lee, Ee-Peng Lim, Mukesh K. Mohania, Yoshifumi Masunaga (Eds.): *Advances in Database Technologies*, ER '98 Workshops on Data Warehousing and Data Mining, Mobile Data Access, and Collaborative Work Support and Spatio-Temporal Data Management, Singapore, November 19-20, 1998.

[Sof98] Softlab Corporation: *Enabler 2.0 Administrator Guide*, Softlab Corporation, Munich, Germany, 1998.

[Teo94] T.J. Teorey: *Database Modeling and Design*, 2nd edition, Morgan Kaufmann 1994.

[Tre95] M. Tresch: *Evolution in Objekt-Datenbanken*, Teubner, Stuttgart, 1995

[Ulb99] T. Ulbricht: *Praktischer Vergleich der multidimensionalen Datenmodelle mehrerer OLAP Produkte.* Internship, FORWISS, Munich, Germany, December 1999.

[Vas98] Panos Vassiliadis: *Modeling Multidimensional Databases, Cubes and Cube Operations*. Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM), Capri, Italy, July 1998

[Vet99] M. Vetterling: *Entwicklung einer automatisierten Schema-Evolutions-Komponente für Informix MetaCube*. Internship, FORWISS, Munich, Germany, December 1999.

[Vos94] G. Vossen: *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*. 2nd Edition, Addison-Wesley, Bonn, Germany, 1994.

[VS99] P. Vassiliadis, T. Sellis: *A Survey of Logical Models for OLAP Databases*. SIGMOD Record, Volume 28, Number 4, December 1999.

[Wid95] J. Widom: *Research Problem in Data Warehousing*. Proceedings of 4th International Conference on Information and Knowledge Management (CIKM), Baltimore, Maryland, USA, November 1995.

[ZaMe82] C. Zaniolo, M.A. Melkanoff: *A Formal Approach to the Definition and the Design of Conceptual Schemata for Database Systems*. ACM Transactions on Database Systems, Vol. 7, No. 1, March 1982

[Zha99] X. Zhang: *Data Warehouse Maintenance Under Interleaved Schema and Data Updates*. Master thesis, Worcester Polytechnic Institute, USA, May 1999.

[ZR98] X. Zhang, E.A. Rundensteiner: *Data Warehouse Maintenance Under Concurrent Schema and Data Updates*. Technical Report WPI-CS-TR-99-8, Worces-

ter Polytechnic Institute, August 1998.

[ZR99] X. Zhang, E.A. Rundensteiner: *The SDCC Framework for Integrating Existing Algorithms for Diverse Data Warehouse Maintenance Tasks*. Proceedings International Database Engineering and Applications Symposium (IDEAS), August 1999, Montreal, Canada.

[ZRD99] X. Zhang, E.A. Rundensteiner, L.Ding: *PSWEEP: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources*. Technical Report WPI-CS-TR-99-14, Worcester Polytechnic Institute, May 1999.

Indexes

Table of figures

Table of definitions

Table of theorems

Table of proofs