JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Sozial- und Wirtschaftswissenschaftliche
Fakultät

Extending data warehouses with hetero-homogeneous dimension hierarchies and cubes

A proof-of-concept prototype in Oracle

(Revision: 1 March 2010)

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften
(Mag.rer.soc.oec)

im Diplomstudium

Wirtschaftsinformatik

Eingereicht von:
Christoph Schütz

Angefertigt am:
Institut für Wirtschaftsinformatik – Data & Knowledge Engineering

Beurteilung:
o.Univ.-Prof. Dr. Michael Schrefl

Mitwirkung:
Mag. Bernd Neumayr

Linz, Februar 2010 (aktualisiert im März 2010)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit mit dem Titel „Extending data warehouses with hetero-homogeneous dimension hierarchies and cubes – A proof-of-concept prototype in Oracle" selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, im Februar 2010 *Christoph Schütz*

Abstract

Data warehouses integrate data from various, mostly heterogeneous sources to provide business executives with the necessary information for a founded strategic analysis. A common issue in data warehousing is the design of the extract, transform, and load (ETL) process which serves as a link between operational databases and the corporate data warehouse. Before the data can be loaded into the data warehouse, the heterogeneous, conflicting schemata need to be reconciled. The schema conflicts are resolved during the transformation phase of the ETL process. The integration of the heterogeneous data sources is arguably the most complex task in managing data warehouses. Furthermore, the elimination of heterogeneities to a certain extent presents a loss of information that might have been useful to the analyst.

Current data warehouse modeling and implementation techniques cannot satisfactorily represent heterogeneities. In order to solve the transformation issue and preserve useful information from the data sources, a novel modeling approach has been introduced in [NST10]. This approach applies multi-level modeling techniques on data warehouses in order to model hetero-homogeneous dimension hierarchies and cubes. For this end, the concepts of m-objects and m-relationships were adapted and the notion of the m-cube was introduced. The thus obtained dimension hierarchies and cubes are homogeneous with respect to a basic common structure shared by all cells of the OLAP cube. The cubes are heterogeneous with respect to dimension and non-dimension attributes. Different sub-cubes may have additional aggregation levels and move measures to more specific granularities (mixed granularities). Existing measures therefore can be changed with respect to their granularity and measure unit; new measures can be introduced to sub-cubes. Dimensions can have additional non-dimension attributes and levels for different cells in the cube.

This thesis presents a proof-of-concept prototype that extends the Oracle database with the capabilities to cope with hetero-homogeneous dimension hierarchies and cubes. M-objects, m-relationships, and m-cubes are implemented in PL/SQL using Oracle's object-relational features, following and refining an implementation concept originally conceived in [Neu10]. The data warehouse developer is provided with a front-end that greatly facilitates the integration of heterogeneous data. A novel logical structure based on object-relational database concepts – better suited for representing heterogeneous data than traditional ROLAP schemata – is employed.

Kurzfassung

Ein Data-Warehouse integriert Daten von verschiedenen, meist heterogenen Quellen um Entscheidungsträger optimal mit Informationen für eine fundierte Analyse zu versorgen. Ein häufiges Problem im Bereich des Data Warehousing stellt die Gestaltung des Extraktions-, Transformations- und Ladeprozesses (ETL-Prozess) dar. Der ETL-Prozess stellt die Verbindung zwischen den operationalen Datenbanken und dem Data-Warehouse dar. Bevor die Daten in das Data-Warehouse geladen werden können, müssen die heterogenen, auseinander laufenden Schemata zusammengeführt werden. Diese Schemakonflikte werden in der Transformationsphase des ETL-Prozesses gelöst. Die Integration der heterogenen Datenquellen ist die vielleicht schwierigste Aufgabe bei der Verwaltung von Data-Warehouses. Darüber hinaus stellt die Bereinigung von Heterogenitäten einen Verlust an möglicherweise nützlicher Information dar.

Heutige Data-Warehouse Modelle und Implementierungstechniken sind nur unzureichend geeignet für die Darstellung von Heterogenitäten. Um das Transformationsproblem zu lösen und in den Datenquellen vorhandene Informationen bestmöglich zu erhalten wurde in [NST10] ein neuer Modellierungsansatz vorgestellt. Dieser Ansatz adaptiert Multi-Level Modellierungstechniken für den Einsatz in Data-Warehouses um hetero-homogene Dimensionshierarchien und Cubes darzustellen. Um dies zu erreichen wurden M-Objects und M-Relationships angepasst und der M-Cube eingeführt. Die so erhaltenen Dimensionshierarchien und Cubes sind homogen in Bezug auf eine grundlegende, gemeinsame Struktur für alle Sub-Cubes. Sie sind heterogen in Bezug auf dimensionale und nicht-dimensionale Attribute. Verschiedene Sub-Cubes können zusätzliche Aggregierungsstufen besitzen und die Granularität von Kennzahlen erhöhen. Bestehende Kennzahlen können verändert werden in Bezug auf die Granularität und die Maßeinheit. Neue Kennzahlen können für bestimmte Sub-Cubes eingeführt werden. Dimensionen können verschiedene nicht-dimensionale Attribute und Ebenen für verschiedene Zellen des Cubes haben.

Die vorliegende Arbeit präsentiert einen experimentellen Prototyp, der die Oracle Datenbank um Fähigkeiten zur Verwaltung von hetero-homogenen Dimensionshierarchien und Cubes erweitert. M-Objects, M-Relationships und M-Cubes sind in PL/SQL implementiert. Der Prototyp nützt dabei die objekt-relationalen Fähigkeiten von Oracle. Dabei wird einem Implementierungskonzept gefolgt, welches ursprünglich in [Neu10] entwickelt wurde. Dem Data-Warehouse-Entwickler wird damit ein Frontend zur Verfügung gestellt welches die Integration von heterogenen Daten erleichtert. Ein neuartiges logisches Datenmodell, basierend auf objekt-relationalen Datenbankkonzepten, wird dazu eingesetzt. Dieses Datenmodell ist besser zur Darstellung von heterogenen Daten geeignet als derzeit eingesetzte ROLAP-Schemata.

Contents

List of Figures

List of Tables

Listings

1 State of the art

This chapter gives an overview of the techniques and technologies that are employed by the hetero-homogeneous data warehouse extension package for the Oracle database. An introduction into the basic concepts of data warehousing is given. More advanced issues like performance considerations are further explained and current OLAP research trends are indicated. The data warehouse extension package is implemented using the Oracle object-relational database. Object-relational concepts are therefore introduced to the reader. An overview of multi-level modeling techniques concludes the chapter.

1.1 Data warehousing

Data warehouses are designed to support executive decisions. As opposed to operational databases that are employed in the daily business, data warehouses are concerned with providing data for strategic analysis. Data warehouses are scalable databases optimized for strategic analysis and decision support; they are characterized by the following attributes (cf. [Inm02], p. 31 et seq.):

- *Subject-oriented*: Whereas operational data tends to be application-oriented, the data in data warehouses is centered around a company's strategic subjects. In business terms, data warehouses manage strategic data, reflecting the company's *core competencies*, as opposed to operational databases which are concerned with *every-day business*, i.e., the products resulting from the company's *core competencies*.

- *Integrated*: The data within the data warehouse comes from multiple, heterogeneous sources. These sources are often conflicting in terms of encoding, measure units, etc.; the conflicts need to be resolved during the process of loading the operational data into the data warehouse. The heterogeneous data are integrated into the data warehouse.

- *Nonvolatile*: Operational databases are designed to perform update operations on the contained data frequently. On the other hand, the data in data warehouses is not changed once it has been loaded into the database.

- *Time variant*: Operational databases reflect the current business reality of a company. When the reality changes, obsolete data is replaced, disregarding historical snapshots. Data warehouses are time variant; they manage historical data which allows the decision maker to track the evolution over time. For data in the data warehouse, the system keeps track of the time period within the data was correct.

The term *online analytical processing* (OLAP) is closely linked to data warehousing; sometimes, the terms *OLAP* and *data warehousing* are even used synonymously. OLAP amounts to the activity of analyzing the complex contents of a data warehouse and stands opposed to *online transactional processing* (OLTP) (cf. [EN07], p. 977 et seq., [Inm02]). OLTP describes the operational tasks supported by 'traditional' database management systems (DBMS) and is diametrically different from OLAP applications. Operational OLTP databases are characterized by frequent update operations, and accurately reflect the current business situation of a company (cf. [EN07], p. 978). Data warehouses, on the other hand, are less concerned with updating data sets once loaded into the database; they rather focus on the analysis of a massive data stock. Consequently, whereas OLTP applications are optimized with respect to frequent update operations, redundancy-free data, and space efficiency, OLAP strives for high performance in retrieving and analyzing huge amounts of data (cf. [Inm02], [JLVV03]).

The generic architecture for data warehouses as presented in [JLVV03] is shown in figure 1.1. At the heart of any data warehouse infrastructure lies the *corporate data warehouse*. The basis for the corporate data warehouse is laid by the operational database systems which serve as the data source. The source schemata are the starting point for the conceptual model of the data warehouse (see section 1.1.1). The conceptual model is then translated into the logical *data warehouse schema* (see section 1.1.2). The various source schemata within the operational database are usually heterogeneous. The data need to be reconciled and integrated. The *extract, transform, and load* (ETL) process is necessary in order to integrate heterogeneous data from various sources into the central data warehouse; it is an important and complex task (see section 1.1.3). Data warehouses need to be scalable. Query performance is an important issue in a data warehouse with huge amounts of contained data. Metadata may be included within the data warehouse in order to improve its performance. Metadata is data about other data; it is usually incorporated in data warehouses in order to more efficiently process information by having available a detailed description of the warehouse's data (cf. [PCTM02], [PCTM03]). The corporate data warehouse contains the strategic data for the whole company. On the other hand, data marts represent small fractions of the data warehouse tailored to the end-user's specific domain.

The data warehouse environment can be described using the *information supply chain* (cf. [KR02], [PCTM03]). The *information supply chain* visualizes the data flow within a data warehouse environment. The various data sources are accessed; its data is extracted, transformed, and loaded into the data warehouse. Data is further aggregated and stored in data marts. The end-user then further processes the data contained in the data marts. The term *information supply chain* is a deliberate analogy to supply chains of manufactured goods; data is stored within a warehouse until they are needed by the client – much like manufactured goods (cf. [PCTM03], p. 12). A formal framework for OLAP applications, including definitions of query operations and consistency criteria for cubes, is provided in [LT09].

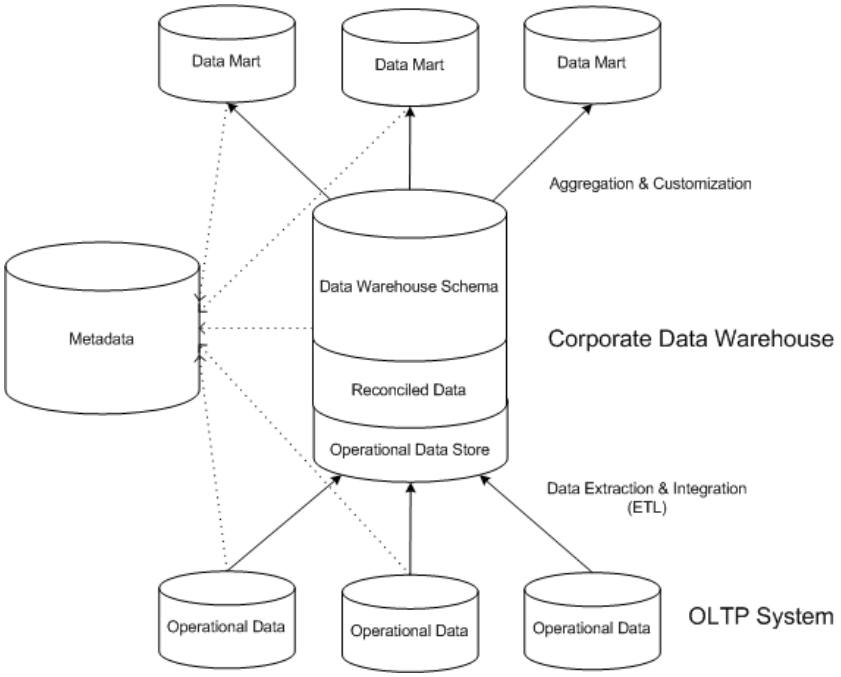


Figure 1.1: A generic data warehouse architecture [JLVV03], p. 3

1.1.1 Conceptual data warehouse modeling

Data warehouses are multi-dimensional databases. The data warehouse contains *facts* of interest for the analysis which are further described by measures [GMR98]. These facts are grouped within multiple dimensions. The visual model that is usually associated with data warehouses is the *cube*. Figure 1.2 depicts a three-dimensional *sales* cube. The example cube contains the *sales* fact which is described by measure *revenue*. The facts of this cube have a *product*, *time*, and *location* dimension. Consequently, each cell of the cube represents a *sale* of a particular product at a specific location in a given time period.

The dimension hierarchy, the facts of interest, and the measures are identified from the source schemata. The conceptual model of an operational database can serve as the source for conceptually modeling the data warehouse. The *Dimensional Fact Model* (DFM) introduced an algorithmic procedure describing how a conceptual data warehouse model can be obtained from an operational source schema [GMR98]. The result of the procedure is a conceptual model with facts, measures, and dimensions. The dimensions are characterized by aggregation levels that are all ordered within a level-hierarchy. Consider a simple operational database schema as depicted in figure 1.3. Starting from the *Order* entity, three dimensions may be identified. The data warehouse architect may choose not to include certain information, e.g., information about the employee that handled the sale or stock information. Figure 1.4 then depicts the visual representation of the conceptual model that describes the previously introduced example *sales* cube;

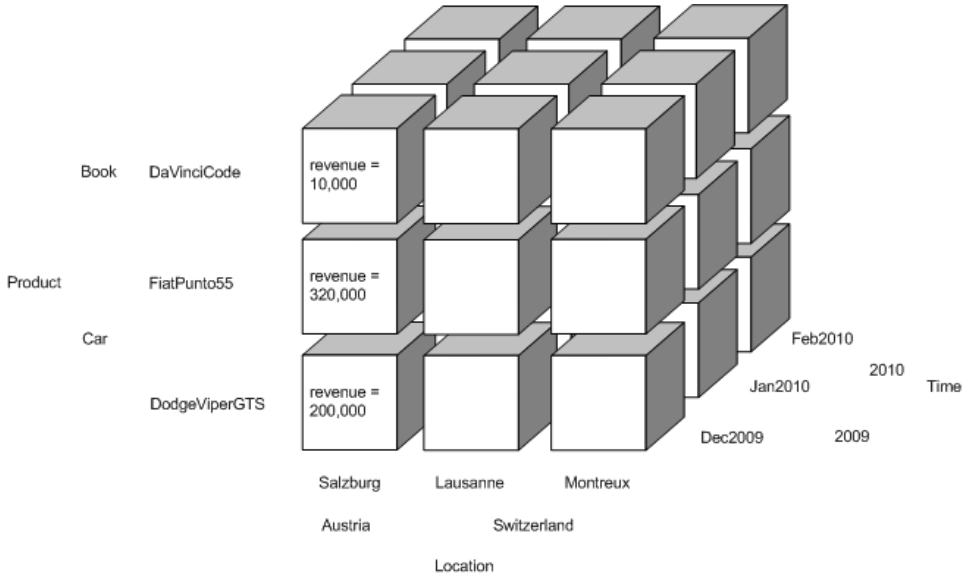


Figure 1.2: A three-dimensional OLAP *sales* cube

information on the notation can be found in [GMR98]. The *sales* fact table is described only by measure revenue. For each dimension, an aggregation level-hierarchy is defined. *Non-dimension attributes* in the DFM provide additional information for a particular level. These attributes cannot be used for aggregation, but may be used for selecting sub-cubes. For example, the number of a city's *inhabitants* is stored as non-dimensional information within the data warehouse; it cannot be used to aggregate cells. However, it may be used as selection criterion, e.g., when selecting a sub-cube of sales in cities with more than 100,000 inhabitants. Another conceptual modeling technique is *Application Design for Processing Technologies* (ADAPT) that has first been introduced in [Bul96].

The central concept when talking about data warehouses and cubes is *granularity*. The granularity of the facts describes their level of detail and is one of the most important aspects of data warehousing (cf. [Inm02], p. 43 et seq.). In OLTP databases, the data is given at a high level of granularity; this is not necessarily true for data warehouses. The operational schema depicted in figure 1.3 that served as the source for the sample cube represents sales events by the *Order* entity. In the table corresponding to this entity, every single sale of a particular product is recorded as a separate tuple containing information about the products that have been sold, the total price, as well as a timestamp. The granularity is thus very high. On the other hand, the data warehouse has already aggregated the granular data to a certain extent. The cube represents the sales at a lower level of granularity in that sales are tracked by product, month, and city. The data warehouse, unlike the operational database, does not allow to retrieve data about single transactions and does not contain information about the employee that operated the counter nor are stocked quantities available. Cube operations may be applied to further aggregate the cells. The dimensions are hierarchically ordered. For instance, a decision

maker may be interested in last year's car sales of the whole country. The aggregation of detailed values to obtain a generalized view at a less specific granularity level is called a *roll-up* operation; the specialization of aggregated facts is called a *drill-down* operation [EN07], [Tha00].

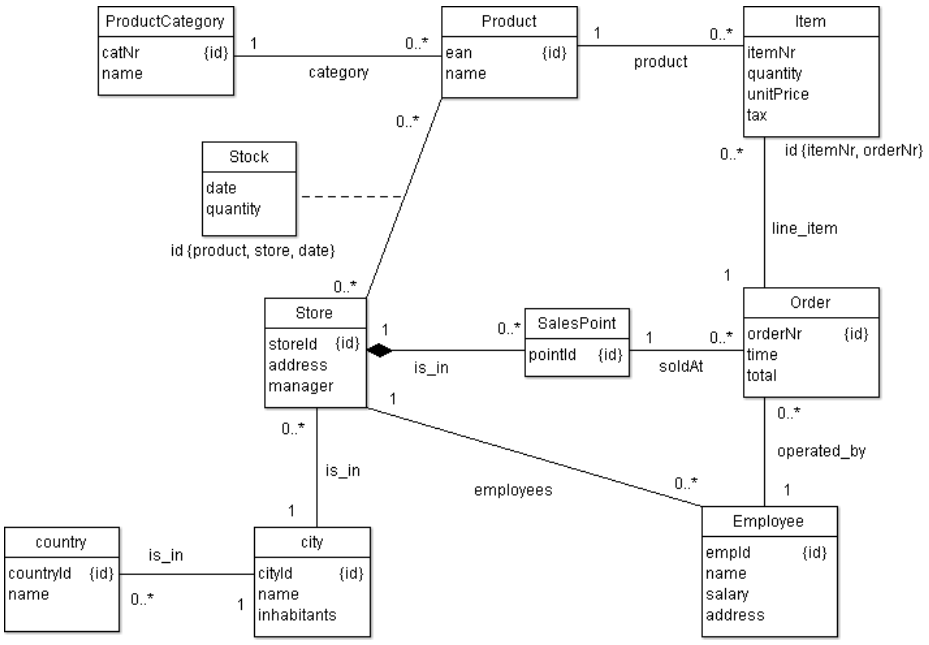


Figure 1.3: An operational database schema (inspired in parts by an example from [Ber08])

Different aggregation functions may be available for different measures. In the given example, measure *revenue* can be summed up to obtain the total revenue at a more specific level of granularity. The analyst can also calculate the average revenue at a specific aggregation level. The aggregation functions that are available for a particular measure within a specific dimension are graphically represented using a dotted arc (figure 1.4). A formal characterization of aggregation functions in data warehouse systems is given by [LT01].

1.1.2 Relational online analytical processing (ROLAP)

Data warehouses can be built using relational databases which is also referred to as relational online analytical processing (ROLAP). Relational tables are used to store measures and non-dimensional attributes. The ROLAP approach is very popular due to the widespread availability of relational databases. A ROLAP data warehouse can be built in any relational database system. Extensions to the SQL standard may by provided in order to better support OLAP applications. For example, the Oracle database allows to define and specify dimension objects and aggregation hierarchies. Furthermore, additional SQL query operators are available in the Oracle database.

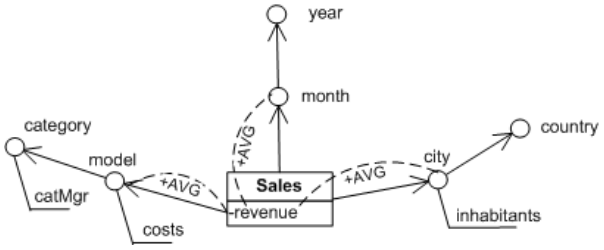
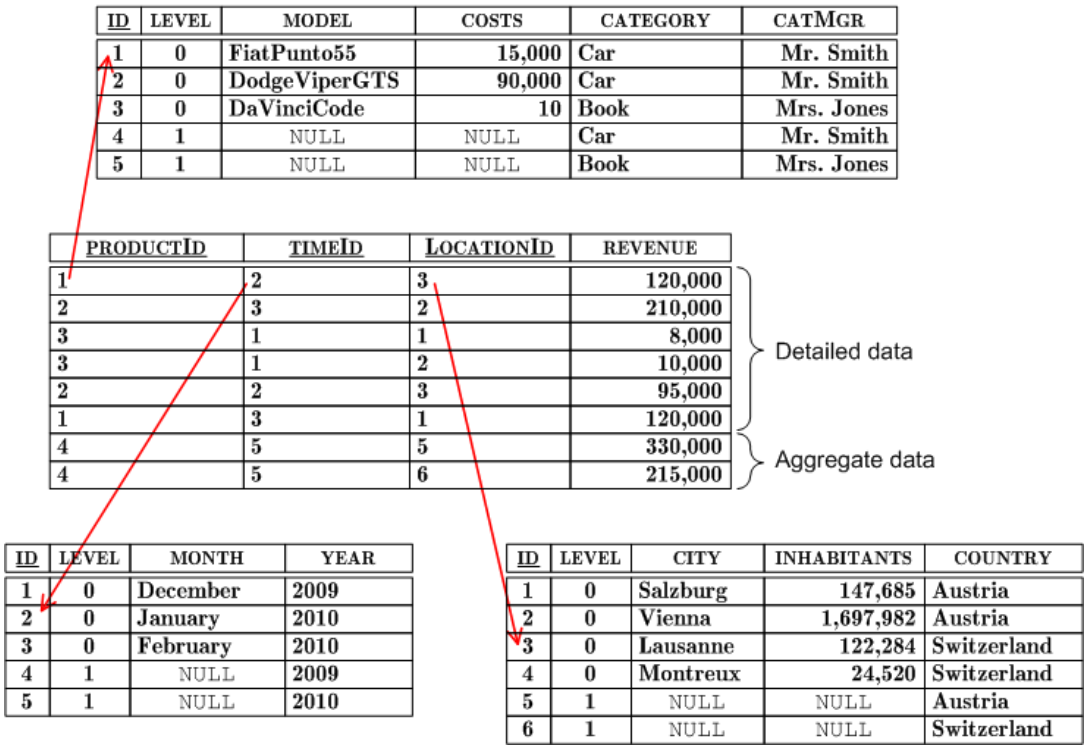


Figure 1.4: A conceptual model of a three-dimensional *sales* cube (example taken from [NST10])

The most common ROLAP schemata are the *star* schema and its variants. The star schema data warehouse organizes measures within a fact table and stores non-dimensional information in denormalized dimension tables (cf. [Lan09], [Hah06], [EN07]). For each dimension of the cube, a separate table is created. Each level of a particular dimension corresponds to a column in the dimension's dimension table. An additional column is introduced for every non-dimension attribute. The fact table references the entries in the dimension tables. An example ROLAP star is illustrated in figure 1.5. The center table is the fact table of the *sales* fact which stores the values for meausre *revenue*. The dimension tables store additional information that is not used for aggregation but may be useful to know in a data warehouse as well, e.g., the category manager or the population of a city. The tuples in the sample dimension tables are identified by surrogate keys which are referenced by the fact table's tuples. The first line in the fact table signifies the sales of *FiatPunto55* in Lausanne in January 2010. The foreign key attributes form the composite primary key of the fact table. A tuple in the fact table corresponds to a cell in the *sales* cube.

The denormalization of the dimension tables entails redundant data in the star schema. The values of non-dimension attributes of a particular aggregation level are redundantly stored. For example, the manager of a particular product category is stored in every tuple. In general, the redundant data are only a minor problem in data warehouses as opposed to OLTP applications where redundancies are a major source of errors. Frequent updates in operational databases are error-prone and slow when data is not normalized (cf. [Cod90]). Data warehouses, on the other hand, are commonly not updated very frequently; redundant data in dimension tables is thus acceptable.

The fact table of star schema organization does not necesseraly store only measures at the most detailed level of granularity; it can be adapted to store aggregate data as well. It might be beneficial to store aggregate data in fact tables in addition to granular data. An additional level column can be added to the dimension tables so that it is possible to store aggregated values (cf. [Hah06]). In the level column, the aggregation level of a particular entry is indicated (see figure 1.5). Non-dimension attributes that are not available for the level indicated in the level column are left empty (`NULL`). This approach introduces additional redundancy, but can improve performance significantly when certain aggregates are queried frequently.



ID	LEVEL	MODEL	COSTS	CATEGORY	CATMGR
1	0	FiatPunto55	15,000	Car	Mr. Smith
2	0	DodgeViperGTS	90,000	Car	Mr. Smith
3	0	DaVinciCode	10	Book	Mrs. Jones
4	1	NULL	NULL	Car	Mr. Smith
5	1	NULL	NULL	Book	Mrs. Jones

PRODUCTID	TIMEID	LOCATIONID	REVENUE
1	2	3	120,000
2	3	2	210,000
3	1	1	8,000
3	1	2	10,000
2	2	3	95,000
1	3	1	120,000
4	5	5	330,000
4	5	6	215,000

ID	LEVEL	MONTH	YEAR
1	0	December	2009
2	0	January	2010
3	0	February	2010
4	1	NULL	2009
5	1	NULL	2010

ID	LEVEL	CITY	INHABITANTS	COUNTRY
1	0	Salzburg	147,685	Austria
2	0	Vienna	1,697,982	Austria
3	0	Lausanne	122,284	Switzerland
4	0	Montreux	24,520	Switzerland
5	1	NULL	NULL	Austria
6	1	NULL	NULL	Switzerland

Figure 1.5: Sample star schema organization

The *fact constellation* and the *snowflake* schema are variations of the star schema. The fact constellation schema introduces for each aggregation level combination a separate fact table where aggregate data can be stored [Hah06]. The snowflake schema, on the other hand, was conceived to overcome the redundancy issue. The dimension tables are normalized; instead of one table per dimension, the tables are arranged within a hierarchy, with one table per dimension and aggregation level (cf. [Hah06], [Lan09], [EN07]).

In ROLAP data warehouses, indexes can be defined on the fact and dimension tables in order to improve performance. *Bitmap* indexes are generally preferred over *B-tree* indexes as the requirements in terms of disk space are usually too much for OLAP applications [Lan09]. Relational OLAP requires the dimension and fact tables to be joined. The performance of these *star joins* can be improved using *bitmap join* indexes [Lan09], [CA09].

A star schema can be generated automatically from entity-relationship diagrams. A recently developed tool, SAMSTAR, based on the semi-automatic method of the same name, can be used to automate this process (cf. [SKA+07], [SKA+08]).

1.1.3 Extract, transform, and load (ETL)

The various operational data sources need to be integrated into a single data warehouse. The issue of data integration is the most important problem in data warehousing [Inm02]. The integration of the heterogeneous source data is an important research topic in the field of data warehousing. Various approaches deal with the issue of integrating heterogeneous data sources, e.g., *schema integration*, *virtual data integration*, and *materialized data integration* (cf. [JLVV03], p. 27 et seq.). *Schema integration* takes the source schemata and produces a single, homogeneous target schema. The *virtual data integration* approach has the data source queried using the views defined during the process of integration. In the *materialized data integration* process, views are defined on the source data with the resulting data sets stored redundantly in *materialized views* in order to boost query performance.

The general ETL process consists of extracting information from the operational sources, applying often complex transformations on the obtained heterogeneous data, and load the transformed input into the global data warehouse. The transformation of heterogeneous data is a major issue in the ETL process: Multiple input sources containing the same or similar information need to be merged, errors in the extracted data need to be corrected, data formats need to be converted, adding a historical dimension to the current-valued input, etc., during the transformation phase (cf. [Inm02], p. 118 et seq.). Performance considerations play a major role as well. Performance issues concerning the ETL process are researched in [TVS07]. Various physical implementations of the ETL flow are evaluated in order to find out the best performing approach; experimental results are included.

An intermediate storage layer – the *operational data store* (ODS) – can be placed between the operational data sources and the global data warehouse to hold "subject-oriented, collectively integrated, volatile, current valued, and detailed data" [JLVV03], p. 4. The ODS in many ways resembles an 'actual' data warehouse. However, some

characteristics of data warehouses are absent, notably the distinctiveness of non-volatility – data in data warehouses is usually stable – and the availability of historical values.

A wide variety of approaches has been developed for mastering the complexity of the ETL process. A UML-based modeling approach is presented in [TLM03]. Recent publications propose the use of ontologies in ETL processes. An approach using *OWL/RDF* for data warehouse design and source data extraction is presented in [NN09]. Another approach uses *topic maps* to develop a multi-dimensional model for OLAP [BTLM01]. Further research has been conducted in the context of knowledge management, where semantic web technologies are used to integrate the structured data in data warehouses with unstructured documents, thus building an *enterprise knowledge portal* [PP03]. A different, yet related, application of semantic web technologies in the context of ETL is presented in [SSC08]. The authors investigate the possibility of automatically generating textual reports from underlying ontology-based ETL models towards a better understandability of the process.

1.2 Object-relational databases

The relational model for database management is well-known and widely accepted. The central entities in relational databases are relations (or tables). The data is organized in tables that follow a defined schema. The user defines a table's columns and their data types upon the creation of the table. A primary key unambiguously identifies a tuple within a table. Current database management systems usually allow the user to alter a table's schema at run-time. Declarative languages are generally used to query the database, e.g., SQL. An extensive source of information about the relational model is found in [Cod90].

With the rise of object-oriented programming, the concept of *object databases* (or object-oriented databases) has been developed. The central entities in object databases are objects. Objects unlike relations not only contain attributes but also incorporate program logic within functions and procedures (methods). The schema of an object is defined by its class (or object type). A system-generated *object identifier* (OID) is assigned to objects in order to provide an unambiguous identity. Initialization tasks are usually assumed by *constructor functions*. Object types can be hierarchically organized, i.e., an object type can have several subtypes. Methods and attributes are inherited by these subtypes. A brief introduction to object databases can be found in [EN07].

Object-relational databases are a hybrid type of database management systems, enhancing relational databases with object-oriented features. Two central entities exist within an object-relational database, namely tables and objects. These two concepts are inter-connected with each other in several ways. An introduction to object-relational database concepts and techniques can be found in [EN07] and [Bro01]. In object-relational tables, user-defined data types can be used as column types. In the Oracle database, these columns are called *object columns* [Ora09]. Object types further serve as patterns for a special kind of tables, namely *object tables*. Each tuple in such an object table corresponds to an instance of a particular, user-defined type [Ora09]. Therefore,

tables in object-relational databases can be typed, which extends the concept of inheritance on tables (cf. [Bro01], p. 15). Nested tables can be defined to hold multiple objects; nested tables allow to store tables within tables. Besides the Oracle database, various other object-relational database management systems are availabe, notably the open source PostgreSQL database [Pos09], Informix [Bro01], DB2 [IBM].

Object-relational tables distinguish between objects and object references. Objects and object references alike can be stored in table columns. An object can be retrieved from an object table using the `VALUE` operator in the SQL `SELECT` clause. The SQL `REF` operator can be used to obtain an object reference from an object table. The PL/SQL counterpart to the `VALUE` operator is provided by the `dbms_sql` package; the `select_object` takes a reference as argument and returns the corresponding object. Notice that the Oracle database returns a copy of the original object. Changes to the object's attributes have to be explicitly stored into the database in order to become persistent. The Oracle database uses typed references, i.e., a reference variable cannot point to objects of any type. Object references are optionally scoped, i.e., the table where the underlying object is stored can be indicated in order to reduce the required storage space. An extensive documentation of Oracle object types can be found in [Ora09].

The possibility of defining functions and procedures requires the existence of some kind of procedural language in addition to the declarative query language. PL/SQL – a procedural extension for SQL – is used in Oracle for this end. Other database management systems provide similar languages, e.g., PL/pgSQL in the PostgreSQL database [Pos09], SQL PL in DB2 [IBM]. In Oracle, any stored procedure or package can be invoked from object type methods. A stored procedure is a piece of program logic stored on-line in the database. The Oracle database alternatively provides the possibility to implement stored procedures in Java [AK09]. The db4o object database, on the other hand, is entirely based on Java and thus its methods are implemented completely in this language (cf. [Ver]).

Query semantics in object-relational databases are distinct from relational databases. Three types of query semantics are to be distinguished in object-relational databases [SS91]. First, queries with relational semantics return relations of data values as results. In this case, values are returned rather than objects. Second, queries can be object-generating such that, when retrieving objects from the database, a copy of these objects is returned by the database system. The database system thus creates a new object that corresponds to the original contained within the database. On the other hand, queries can be object-preserving, i.e., references to the original objects are returned.

An example from the Oracle database can serve to illustrate the difference between object-preserving and object-generating queries. In the PL/SQL language, the `select_object` method of the `utl_ref` package is implemented with object-generating semantics: Only a copy of the database object is returned and the `update_object` needs to be called in order to persist changes [Ora09]. The `select_object` method corresponds to using the `VALUE` operator in SQL statements. The SQL `REF` operator possesses object-preserving query semantics: A reference to the object is returned rather than a copy.

1.3 Multi-level modeling

The well-known object-oriented model distinguishes between the *class* or *type* level and the *object* or *instance* level. The class describes the properties for the entities at the *object* level. Classes can be placed in concretization relationships, thus enabling to define a class hierarchy. Objects belong to particular classes and assert values for the defined attributes. This two-level model has become a pre-vailing programming paradigm in software development. A great magnitude of object-oriented programming languages has been developed. The model has even been extended to database systems (see section 1.2).

The two-level object-oriented model, however, is insufficient for some applications. Entities quite often have a dual nature, possessing the characteristics of objects and classes alike. Consider a *Product* class which is further subtyped into several product categories, namely books and cars. Based on the two-level object model, classes *Book* and *Car* are defined as specializations of the *Product* class, i.e., they are subtypes of *Product*. Entities *Book* and *Car* are classes, i.e., they define a general structure followed by the instantiating objects. Product models may now be created by instantiating these classes and thus obtaining an object of type *Book* or *Car*. For example, a new object *daVinciCode* may be created by instantiating the *Book* class. This may be sufficient for a variety of applications. In some cases, however, an application might demand for a book in general and not for a concrete model. The existence of an object *book* as an instance of class *Product* is required in this case. The notion of *clabjects* expresses this duality of an entity being at the same time a class of objects and an instance of a particular class itself [AK00].

A variety of modeling techniques have been conceived to break the pre-dominance of the two-level object-oriented model. *Power types* represent a concept that is closely related to the notion of *clabjects* (cf. [OF98], [GPHS06]). The *deep instantiation* approach goes a step further in that "an instantiation mechanism is proposed that recognizes and fully supports the class/object duality" [AK01], p. 27/28. Recently, multi-level objects (m-objects) and relationships (m-relationships) were introduced. This approach accounts for the class/object duality and provides a means for modeling classes/objects at various abstraction levels [NGS09]. M-relationships are further introduced in order to represent relationships at various abstraction levels. A detailed comparison of these modeling techniques can be found in [NS08].

A power type is defined as an "object type whose instances are subtypes of another object type" [OF98], p. 28. A meta-modeling framework based on power types has been proposed in [GPHS06]. The concept of *deep instantiation* proposes a novel approach for the instantiation relationship with an inherent support of the dual nature of objects.

2 Hetero-homogeneous dimension hierarchies and cubes

Current data warehouse modeling and implementation approaches are insufficient for representing heterogeneities within dimension hierarchies and cubes. However, the operational source schemata are usually heterogeneous, which requires the data to be transformed in order to obtain a homogeneous data warehouse schema. The issue of heterogeneous source data can be resolved by applying multi-level modeling techniques on data warehouses to introduce the concept of *hetero-homogeneous* cubes. The approach as presented in this chapter was first introduced and formalized in [NST10]. The concept of m-objects and m-relationships is adapted in order to model hetero-homogeneous dimension hierarchies and cubes. The hetero-homogeneous data warehouse is homogeneous with respect to a common base schema, whereas different sub-cubes and dimension branches may introduce additional measures and aggregation levels. This chapter establishes the formal foundations of the implemented proof-of-concept prototype. The definitions presented in this chapter are generally taken from [NST10]. Changes to the original definitions have been necessary with respect to the implementation of the Oracle extension package; the changes made are marked as such.

2.1 Motivation

Data warehouses represent a simplified model of a company's business environment. The purpose of a data warehouse is to support managers with their entrepreneurial decisions. The operational data from day-to-day business serves as the source for the strategic analysis carried out using data warehouses. These operational data sources exist in large numbers; the data warehouse's stock of data is composed from the various operational data sources that exist within the company. OLTP databases with varying schemata commonly serve as the source for the warehouse's data. The thus obtained data is usually heterogeneous, which requires the data to be transformed. During the *transformation* part of the *extract, transform, and load* (ETL) process, heterogeneities in the source data are eliminated. The transformation and integration process is complex and cumbersome, and possibly the most crucial task when building a data warehouse (cf. [Inm02], p. 118). It is certainly not the straight-forward exercise it seems to be at the first glance.

Current data warehousing technologies require the heterogeneous source schemata to be integrated, thus eliminating the heterogeneities and producing a homogeneous data warehouse. Various integration approaches have been conceived, ranging from schema mapping to virtual view approaches (cf. [JLVV03]). The transformation process is a challenging task due to the heterogeneities in the source schemata. Furthermore, when

reconciling data and thus producing a homogeneous data warehouse, valuable information is lost that could have been useful for the decision maker's analysis. Providing a means for representing heterogeneities in data warehouses may ease the transformation issue. Hetero-homogeneous dimension hierarchies and cubes provide a concept to achieve this task.

2.2 Modeling hetero-homogeneous dimension hierarchies with m-objects

M-objects describe real-world entities on multiple levels of abstraction, as opposed to the two-level modeling approach known from traditional object-oriented modeling techniques. While the two-level object model, e.g., in Java and C++, only distinguishes between *class* and *instance* level, the concept of m-objects has various levels of abstraction. Each m-object is at the same time an *instance* representing a particular real-world entity, and the *class* for m-objects at a more specific abstraction level. The multi-level modeling approach using m-objects has been first introduced in [NGS09] and applied on data warehouses in [NST10]. Each m-object describes a hierarchy of abstraction levels (see chapter 1). Its single top-level defines the m-object's level of abstraction. The underlying real-world entity's properties are reflected by an m-object's attributes; the attributes are always defined for a particular level. Values are only provided for attributes defined at the m-object's level of abstraction. Attributes defined at more specific abstraction levels impose a common structure on m-objects at more specific levels of abstraction

A formal definition of m-objects is provided in [NST10]: M-objects are formally defined as a 6-tuple data structure $(L_o, A_o, P_o, l_o, d_o, v_o)$. An m-object o has a set of levels L_o from a universe of levels L which is partially ordered by relation P_o. Relation P_o provides a mapping from levels to parent-levels; it is referred to as the m-object's level-hierarchy. A tuple $(l', l) \in P_o$ signifies that level l' is a sub-level of l, which is alternatively expressed as $l' \prec l$ or equivalently as $l \succ l'$. Note that operators $\prec$ and $\succ$ are transitive. An m-object further has a set of attributes where each attribute has a level and a domain assigned. Function $l_o : A_o \to L_o$ maps a level of the m-object's set of levels to each attribute of m-object o; function $d_o : A_o \to \mathit{datatype}$ maps a data type to an attribute – the attribute's domain. An attribute optionally has a value assigned from its domain; this mapping is provided by function $v_o : A_o \to V$, where V is a universe of values. Each m-object o has a single top-level $\hat{l}_o$. The top-level $\hat{l}_o$ of m-object o is defined as the level $l \in L_o$ where no mapping from level l to a parent-level $l' \in L_o$ exists within the m-object's level-hierarchy P_o.

M-object attributes may further be described by metadata at different meta levels. The formal definition of m-objects may be altered as follows: $(L_o, A_o, P_o, l_o, d_o, mv_o, def_o)$. The original definition of m-objects is thus adapted in order to allow for the representation of attribute metadata. The additional function $mv_o : A_o \times ML \to MV$ maps a meta-value from a universe of meta-values MV onto an attribute of m-object o for a particular meta level from a universe of meta levels ML. The universe of meta levels ML is a subset of the universe of levels L, i.e., $ML \subseteq L$. Function $def_o : A_o \times ML \to \mathbb{B}$ defines for a

given meta level and a specific attribute whether the value assigned by function mv_o is a default value or not. The symbol $\mathbb{B}$ represents the boolean domain. The global function $metalevel : MV \rightarrow ML$ maps a meta level from the universe of meta levels ML onto a meta-value from the universe of meta-values MV. A particular meta-value therefore always belongs to exactly one meta level, i.e., $\forall v \in MV : \exists l \in ML : metalevel(v) = l$. Metalevels may be partially ordered; the order is established by the global relation $P_{metalevel} \subseteq ML \times ML$, mapping a parent-level onto each meta level from the universe of meta levels. This formal definition of functions mv_o and $metalevel$ allows for the use of an m-object hierarchy in order to represent and store metadata about attributes (see chapter 3).

The universe of meta-values MV may be defined as a subset of the universe of values V. Meta-level $\emptyset$ may then be introduced to the universe of meta levels ML, meaning that an assigned value is not metadata. For each value in the universe of values not in the universe of meta-values, its assigned meta level is $\emptyset$, i.e., $\forall v \in V \setminus MV : metalevel(v) = \emptyset$. Function v_o of the original m-object definition is then defined using function mv_o such that their values are equal if the meta level is $\emptyset$ and it is not a default value that is returned by mv_o, i.e., for attribute $a \in A_o$ the return value of $v_o(a) := mv_o(a, l)$ where $l = \emptyset \wedge \neg def(a, l)$. Consequently, function mv_o is used to assign values to attributes. It can further be used to define default values for attributes.

Consider, for instance, products in the real world. M-object *Product* can be used to describe the various abstraction levels of products. A product in the real world can be regarded from different levels. The set of levels of m-object *Product* in figure 2.1 comprises three levels: *top*, *category*, and *model*. The m-object's level-hierarchy is defined by relation $P_{Product} := \{(top, \emptyset), (category, top), (model, category)\}$, thus leaving level *top* as the m-object's top-level. Alternatively, the partial order of these levels is expressed as follows: $top \succ category \succ model$. The level-hierarchy reflects the fact that each product can be seen from different aspects. Attributes are introduced to describe products at different abstraction levels. A category is managed by a category manager – attribute *catMgr* whose domain is *Integer*. Each *model* has an attribute *costs* defined, its domain being *Float*. Notice that m-object *Product* does not define a value for these attributes since all of the introduced attributes were defined for a level other than the m-object's top-level. Attribute metadata at meta level *unit of measurement* may be set for attribute *costs*. Costs are generally measured in *euros*. The unit of measurement of an attribute is depicted using square brackets in figure 2.1. At meta level *quantity*, the meta-value for attribute *costs* is *currency*, with meta level $quantity \succ unit$.

M-objects are arranged in concretization relationships. A concretization of a particular m-object is always defined for a more specific abstraction level than its parent m-object. The concretization of an m-object o is said to be its descendant m-object. The concretizing m-object's top-level has to be a second top-level of its parent m-object, i.e., the direct (not transitive) sub-level of the parent m-object's top-level. The concretizing m-object contains all levels from the parent m-object that are located underneath the concretizing m-object's top-level within the parent's level-hierarchy. An m-object may introduce new levels with respect to its parent m-object. The relative order of the levels defined by the parent m-object, however, has to be persistent. Newly introduced levels must have

parent-levels only in the introducing m-object's level-hierarchy. Attributes are inherited from parent m-objects, i.e., they are contained within the concretizing m-object's set of attributes as well. Multiple concretization is allowed as long as the parent m-objects share a common second top-level. The concretizing m-object's top-level is a direct sub-level of each parent m-object's top-level in their respective level-hierarchies. A *consistent concretization* of m-objects must therefore comply to the following criteria [NST10]:

1. The top-level of the concretizing m-object o' has to be the second top-level of its parent m-object o, i.e., the concretizing m-object's top-level $\hat{l}_{o'}$ is a direct sub-level of the parent m-object's top-level $\hat{l}_o$ in the parent's level-hierarchy: $(\hat{l}_{o'}, \hat{l}_o) \in P_o$. For example, the top-level of m-object *DaVinciCode* in figure 2.1 – level *model* – is a sub-level of level *category* in m-object *Book*.

2. The set of levels of concretizing m-object o' contains every level that is a (transitive) sub-level of $\hat{l}_{o'}$ within the parent m-object's level-hierarchy (*level containment*). M-object *Product* contains levels *top*, *category*, and *model*; the following order is defined: $top \succ category \succ model$. The concretizing m-object *Car* with top-level *category* must contain levels *category* and *model*.

3. Levels shared by two m-objects within a concretization relationship must have the same relative order (*level order compatibility*). Consequently, a level l that is the sub-level of another level l' in the level-hierarchy of m-object o has to be a (transitive) sub-level in the level-hierarchy of the concretizing m-object o' as well.

4. Each level that is newly introduced by a concretizing m-object o' has parent-levels only within the level-hierarchy of o' (*locality of level order*).

5. Attributes of the parent m-object are contained in the concretizing m-object's set of attributes if the associated level is shared by the two m-objects (*attribute containment*). The attributes are said to be inherited by the concretizing m-objects. Notice that inherited m-objects are not shown in the graphical representation of m-objects (figure 2.1). The implementation does not require the inherited attributes to be added explicitly to the concretizing m-objects as they are implicitly contained in the descendant's set of attributes (see chapter 3).

6. Attributes shared by two m-objects within a concretization relationship are defined at the same level (*stability of attribute levels*), have the same domain (*stability of attribute domains*), and have the same value if a value has been defined at the parent m-object (*compatibility of attribute values*).

Metadata about attributes may be changed by concretizing m-objects. Only default meta-values may be changed by concretizing m-objects. Given an m-object o' under parent m-object o, the meta-value for a particular attribute $a \in A_o \cap A_{o'}$ is the same for a given meta level l from the universe of meta levels ML if the attribute meta-value is not a default value. The value may only be altered in case that the value for this level and attribute has been marked as default; it is necessarily the same if it has been marked

as a shared value by parent m-object o, i.e., $mv_o(a,l) = mv_{o'}(a,l)$, if $\neg def_o(a,l)$ where $l \in ML$ and $a \in A_o \cap A_{o'}$.

The multi-level nature of m-objects is well-suited for modeling data warehouse dimensions. Dimensions represent data at various abstraction levels, providing a hierarchical organization of levels. Whereas traditional approaches for modeling cube dimensions are bound to homogeneous hierarchies, m-objects provide a concept that allows for heterogeneities to be represented. M-object hierarchies can be used to represent dimensions within cubes. M-object attributes represent the non-dimensional information, i.e., information that cannot be used for aggregation, the equivalent to *non-dimension attributes* in the *Dimensional Fact Model* [GMR98]. Concretizing m-objects may introduce new levels, thus introducing new aggregation levels for sub-branches of a particular cube dimension. Consequently, heterogeneities can be represented using the m-object modeling approach.

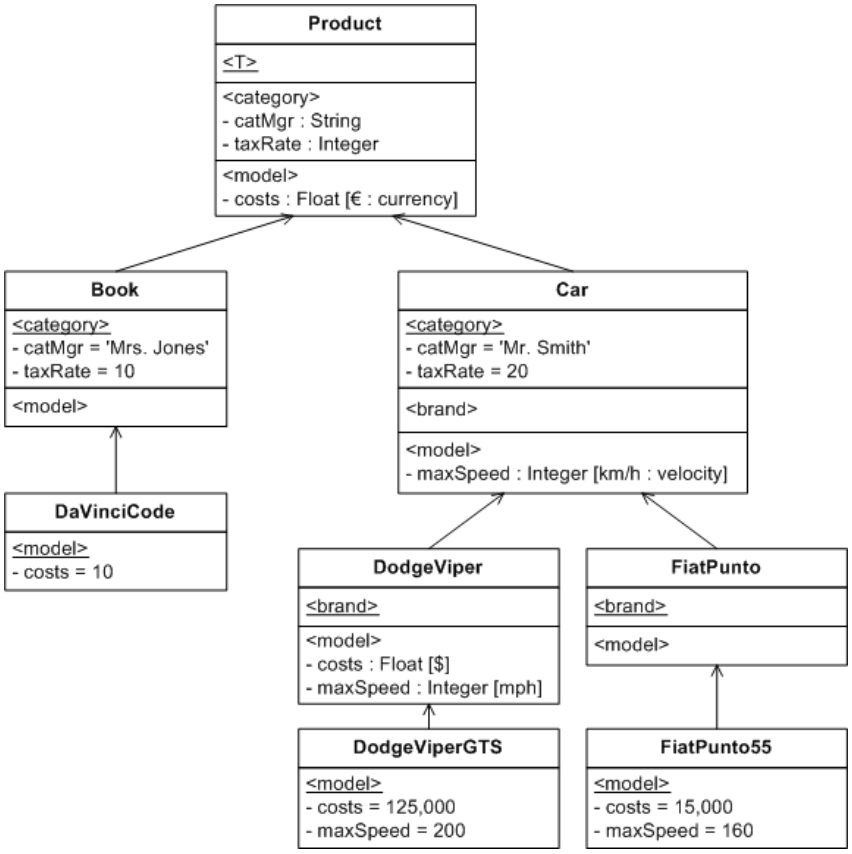


Figure 2.1: The hetero-homogeneous *product* dimension of a *sales* cube modeled with m-objects (extension of the original example as described in [NST10])

Figure 2.1 illustrates how an m-object hierarchy may be used to model the *product* dimension of a *sales* cube. The *product* dimension presents an example where heterogeneities in the dimension hierarchy occur. Different sub-branches of the dimension have

different aggregation paths. The *product* dimension is a hetero-homogeneous dimension. It is homogeneous with respect to a common general structure as defined by the dimension's root m-object *Product*. The *product* dimension generally defines the following aggregation hierarchy established by the root m-object: $top \succ category \succ model$. The *Car* branch introduces an additional level *brand* which is not contained in the *Book* branch of the *product* dimension. The dimension is therefore heterogeneous in the sense that some sub-branches contain additional aggregation levels. Furthermore, different sub-branches contain additional non-dimensional information, i.e., new attributes are introduced for common levels. For instance, m-object *Car* defines the additional attribute *maxSpeed* at level *model*, a level shared with the *Book* branch. However, m-objects under *Book* do not have an attribute *maxSpeed*. The *product* dimension is thus heterogeneous with respect to the non-dimensional information.

A dimension may as well contain heterogeneities with respect to the attribute metadata. For example, attribute *costs* has a meta-value assigned at the *unit* level. The *costs* of a product are generally measured in *euros*; this is defined by m-object *Product* which first introduces the attribute. Metadata are inherited from parent m-objects and may be concretized by descendant m-objects. For instance, cars of brand *FiatPunto* inherit attribute *costs* which has been defined for the *model* level by m-object *Product*. Along with the attribute, its domain and metadata is inherited. M-object *FiatPunto* does not further concretize metadata for the *costs* attribute. Descendant m-objects of *FiatPunto* consequently measure their *costs* in *euros*. M-object *DodgeViper* at the *brand* level, however, changes the unit of attribute *costs*. Being an American car brand mainly sold in the United States, the costs for cars of brand *DodgeViper* are given in *U.S. dollars*.

A consistent dimension modeled with m-objects can be described by a 4-tuple data structure [NST10]. Dimension $D = (O_D, A_D, L_D, H_D)$ consists of a set of m-objects O_D, a set of attributes $A_D = A_{o_1} \cup \ldots \cup A_{o_n}$, a set of levels $L_D = L_{o_1} \cup \ldots \cup L_{o_n}$, and a relation $H_D : O_D \times O_D$ organizing the set of m-objects O_D within a hierarchical order. Relation H_D defines the (direct) parents of an m-object o. An m-object $o' \in O_D$ is said to be a *direct concretization* of an m-object o iff the pair $(o', o) \in H_D$. The concretization of m-objects is a transitive relation, i.e., an m-object o' can be an *indirect concretization* of an m-object o if it is a direct concretization of an m-object $o'' \in O_D$ that is a (direct or indirect) concretization of o. M-object o' is an indirect concretization of o iff $(o', o) \in H_D^+$ and a direct or indirect concretization of o iff $(o', o) \in H_D^*$, with H_D^+ being the *transitive* closure and H_D^* being the *transitive-reflexive* closure of H_D. Direct or indirect concretization relationships may be expressed using $\succ$ and $\prec$, where $o \succ o'$ denotes that m-object o is a direct or indirect ancestor of m-object o, and $o' \prec o$ being the equivalent stating that o' is a direct or indirect concretization of m-object o. The following criteria must be fulfilled by the consistent dimension [NST10]:

1. All m-objects of the dimension are an m-object according to the (extended) definition of m-objects.

2. All pairs of m-objects $(o', o) \in H_D$ are consistent concretizations.

3. Attributes and levels are introduced exactly once, i.e., two m-objects that share a level $l \in L_{o'} \cap L_{o''}$ or an attribute $a \in A_{o'} \cap A_{o''}$ have either a common ancestor that introduces the level l or attribute a, respectively, or are in a concretization relationship:

   a) $a \in A_{o'} \cap A_{o''} : \exists \bar{o} \in O_D : (o', \bar{o}) \in H_D^* \wedge (o'', \bar{o}) \in H_D^* \wedge a \in A_{\bar{o}}$ (*unique induction rule for attributes*)

   b) $l \in L_{o'} \cap L_{o''} : \exists \bar{o} \in O_D : (o', \bar{o}) \in H_D^* \wedge (o'', \bar{o}) \in H_D^* \wedge l \in L_{\bar{o}}$ (*unique induction rule for levels*).

4. An m-object o' with top-level l must be the concretization of an m-object $\hat{o}$ with top-level l' if $o' \prec o$ where $(l, l') \in P_o$. This means that m-object *Salzburg* with top-level *city* which is an indirect concretization of *Location* must concretize m-objects *Alps* and *Austria* with top-levels *region* and *country*, respectively, due to the fact that $(city, region) \in P_{Location}$ and $(city, country) \in P_{Location}$.

The set of levels L_D of dimension D has an implicit partial order due to the *unique induction rule for levels* and the *level order compatibility* rule [NST10]. A level $l' \in L_D$ is a sub-level of $l \in L_D$ written as $l' \prec l$ iff an m-object $o \in O_D$ exists with a level-hierarchy P_o where l' is a (transitive) descendant of l. The partial order can be obtained from the virtual level-hierarchy that can be obtained by forming the union of the level-hierarchies of all m-objects $o \in O_D$. It is not mandatory for two levels of the same m-object to be partially ordered. Two levels of the same m-objects may have no order defined. This allows for the definition of alternative aggregation paths. Levels *country* and *region* of the *location* dimension depicted in figure 2.2 are not ordered.

The *location* dimension depicted in figure 2.2 is an example for a hetero-homogeneous dimension hierarchy with alternative aggregation paths and multiple concretization. The root m-object *Location* defines the general aggregation hierarchy of the dimension. The *city* level is defined by m-object *Location* to be the most specific granularity. Data can either be aggregated on the *country* or the *region* level. Levels *country* and *region* are not in a level to parent-level relationship with each other. M-object *Switzerland* at the *country* level introduces an additional aggregation level, namely *kanton*. The induction of level *kanton* makes the *location* dimension a hetero-homogeneous dimension, as data in other countries, e.g., *Austria*, may not be rolled up to the *kanton* level. Furthermore, m-object *Switzerland* introduces more heterogeneities by defining level *store* under *city*, thus providing data at a more specific level of granularity. Notice that each m-object with top-level *city* must concretize both an m-object with top-level *region* and an m-object with top-level *kanton* or *country*, respectively.

2.3 Modeling hetero-homogeneous cubes with m-relationships

Heterogeneities in the fact schema of a cube can be modeled using the concept of m-relationships. The original definition of m-relationships as presented in [NGS09] is ex-

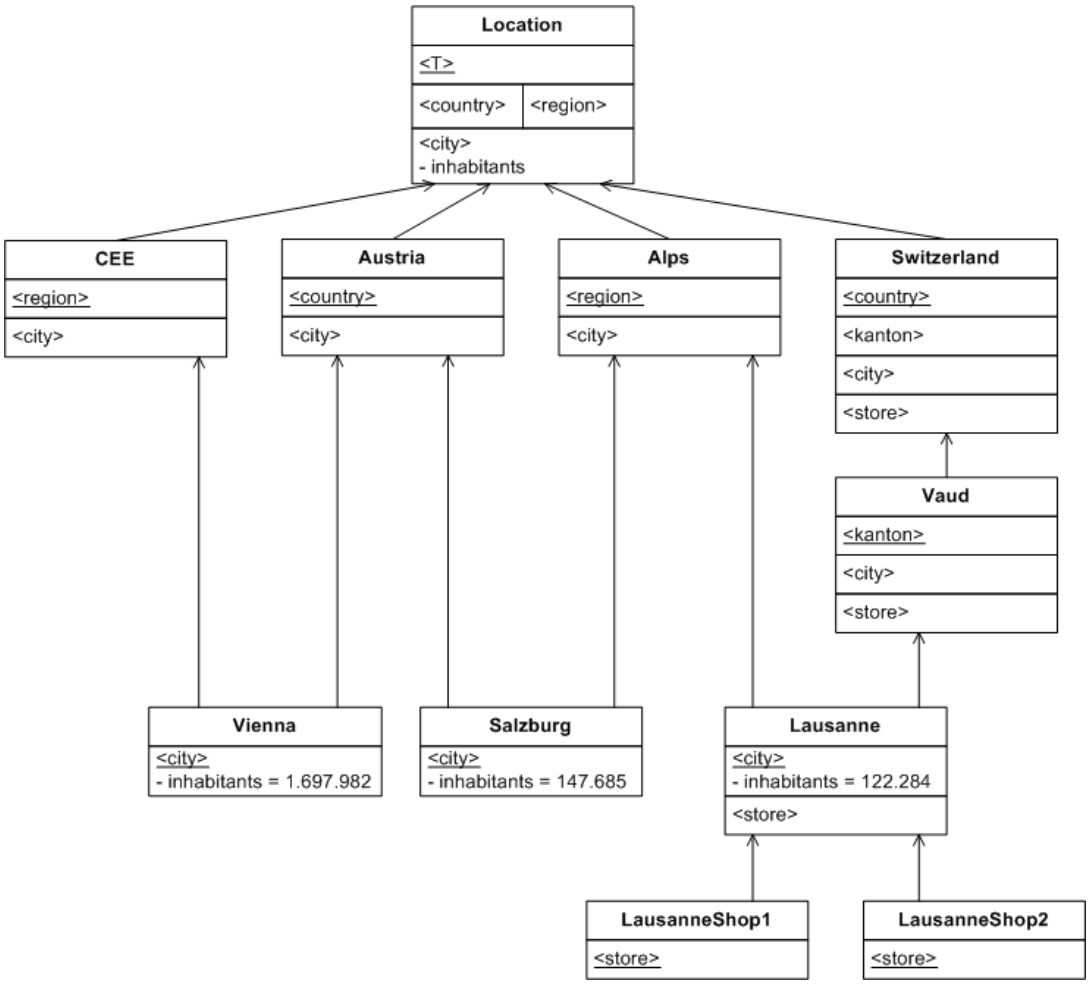


Figure 2.2: The hetero-homogeneous *location* dimension of a *sales* cube modeled with m-objects (adaptation of the original example as described in [NST10])

tended with measures. Heterogeneous facts in terms of measure granularity, measure units, and available measures are supported. Different sub-cubes may thus have additional measures, different levels of granularity, as well as different units for certain measures. The notion of the *m-cube* was then introduced in [NST10] to represent hetero-homogeneous cubes, building on the concept of m-relationships.

2.3.1 Adopting m-relationships for data warehouses

M-relationships are multi-level connections between m-objects. Relationships between m-objects are described at various levels of abstraction. The concept of m-relationships was first introduced in [NGS09] and applied on data warehouses in [NST10]. The original definition of m-relationships was extended with measures in order to be applicable on data warehouses. M-relationships represent the facts of a data warehouse.

M-relationships are described by a 6-tuple data structure [NST10]. The 6-tuple $r = (o_1, \ldots, o_n; M_r, b_r, u_r, f_r, v_r)$ describes an m-relationship r of an n-dimensional cube. The coordinate of r denoted by $coord(r) = o_1, ..., o_n$ denotes the m-objects that are connected by the m-relationship. An m-relationship connects m-objects at various levels of abstraction – the connection-levels. A connection-level denotes a certain level of abstraction of the relationship. For an n-dimensional m-relationship the connection-level l is composed of n levels: one from each of its coordinate's m-objects, i.e., $l \in L_{o_1} \times \ldots \times L_{o_n}$. An m-relationship's top connection-level is defined as the top-level of each m-object that constitute the m-relationship's coordinate, i.e., $\hat{l}_r := (\hat{l}_{o_1}, \ldots, \hat{l}_{o_n})$. The set of measures M_r contains the measures defined for the m-relationship. Each measure has a connection-level defined by function $b_r : M_r \rightarrow (L_{o_1} \times \ldots \times L_{o_n})$. The connection-level defines the level of granularity of the measure. Each measure further has a unit of measurement defined by function $u_r : M \rightarrow U$ where U is a universe of units. A distributive aggregation function is assigned to each measure by function $f_r : M \rightarrow \{\textsc{Sum}, \textsc{Max}, \textsc{Min}\}$. A measure m has an asserted value defined by function $v_r : M \rightarrow V$ iff the connection-level of m is equal to the top connection-level $\hat{l}_r = b_r(m)$. A measure unit $u \in U$ belongs to a measure type $t \in T$ with T being a universe of measure types. The mapping of a measure unit onto a measure type is given by function $type : U \rightarrow T$.

The m-relationship connecting m-objects *Product*, *Time*, and *Location* – i.e., the m-relationship at the cube's root-coordinate – depicted in figure 2.3 introduces measure *revenue* at connection-level $\langle model, month, city \rangle$. The value for this measure is therefore asserted only by the m-relationships at top connection-level $\langle model, month, city \rangle$, namely between m-objects *DaVinciCode*, *Jan2010*, and *Salzburg* and *DaVinciCode*, *Jan2010*, and *Vienna*, respectively, in the example depicted in figure 2.3. The aggregation function for measure *revenue* is SUM, the measure unit €.

The original definition of m-relationships can be adapted to support metadata and default values at various meta levels. The definition of m-relationships may thus be altered to $r = (o_1, \ldots, o_n; M_r, b_r, mv_r, def_r)$, adding functions mv_r and def_r which replace functions f_r, v_r, and u_r. Metadata at various meta levels can be used to represent measure units, types and aggregation functions. Function $mv_r : M \times ML \rightarrow MV$ maps a meta-value from a universe of meta-values MV onto a measure of m-relationship r for a

particular meta level from a universe of meta levels ML. Function $def_r : M \times ML \to \mathbb{B}$ defines for a given meta level and a specific measure whether the value assigned by function mv is a default value or not. The global function $metalevel : MV \to ML$ maps a meta level from the universe of meta levels ML onto a meta-value from the universe of meta-values MV. A particular meta-value therefore always belongs to exactly one meta level. Meta-levels may be partially ordered; the order is established by the global relation $P_{metalevel} \subseteq ML \times ML$, mapping a parent-level onto each meta level from the universe of meta levels. The aggregation functions may also be represented using metadata by introducing a meta level for aggregation functions. As with m-object attribute metadata, an m-object hierarchy may be used to represent measure metadata.

Functions f_r, v_r, and u_r of the original m-relationship definition may be defined using the meta-value function. Meta-level $unit \in ML$ may be used to represent a measure's unit of measurement. Function u_r in the original m-relationship definition can be defined using function mv_r. In this case, the return value of function u_r for a measure m defined by m-relationship r is equal to the return value of function mv_r for the same measure of the same m-relationship at meta level $unit$, i.e., $u_r(m) := mv_r(m, unit)$. Likewise, the type of a unit u for a measure m may be defined to be represented as metadata at meta level $quantity \in ML$, with $type(u) := mv_r(m, quantity)$. The aggregation function of a measure m may be defined at meta level $function \in ML$, with $f_r(m) := mv_r(m, function)$. The universe of measure units U as well as the universe of measure types T may be defined as sub-sets of the universe of meta-values MV, i.e., $U, T \subseteq MV$ where $\forall v \in U : metalevel(v) := unit$ and $\forall v \in T : metalevel(v) := quantity$. In general, the same rules apply for measure default values as for the default values of m-object attributes (see section 2.2).

Similar to m-objects, m-relationships may be concretized. M-relationships and connection-levels are partially ordered. The partial order of m-relationships is crucial for determining whether two m-relationships are in a concretization relationship. The order of m-relationships is defined through the partial order of their coordinates. The order of the coordinates is in turn given by the partial order of the m-objects that constitute the coordinate. For the n-dimensional coordinates $(o'_1, \ldots, o'_n) \in (O_{D_1} \times \ldots \times O_{D_n})$ and $(o_1, \ldots, o_n) \in (O_{D_1} \times \ldots \times O_{D_n})$ the partial order is defined as follows [NST10]:

- Coordinate $(o'_1, \ldots, o'_n)$ is a sub-coordinate of – i.e., a descendant of or equal to – coordinate $(o_1, \ldots, o_n)$, written as $(o'_1, \ldots, o'_n) \preceq (o_1, \ldots, o_n)$, iff for all dimensions, $i = 1..n$, the respective dimension m-object $o'_i \in O_{D_i}$ is a descendant of or equal to m-object $o_i \in O_{D_i}$.

- Coordinate $(o'_1, \ldots, o'_n)$ is a proper sub-coordinate of – i.e., a descendant of – coordinate $(o_1, \ldots, o_n)$, written as $(o'_1, \ldots, o'_n) \prec (o_1, \ldots, o_n)$ iff it is a sub-coordinate according to the previous definition and for at least one dimension m-object $o' \in O_{D_i}$ is a descendant of $o \in O_{D_i}$.

- Coordinate $(o'_1, \ldots, o'_n)$ is said to overlap with coordinate $(o_1, \ldots, o_n)$, written as $(o'_1, \ldots, o'_n) \between (o_1, \ldots, o_n)$ iff for all dimensions, $i = 1..n$, the respective dimension m-object $o_i, o'_i \in O_{D_i}$ are in a concretization relationship or equal.

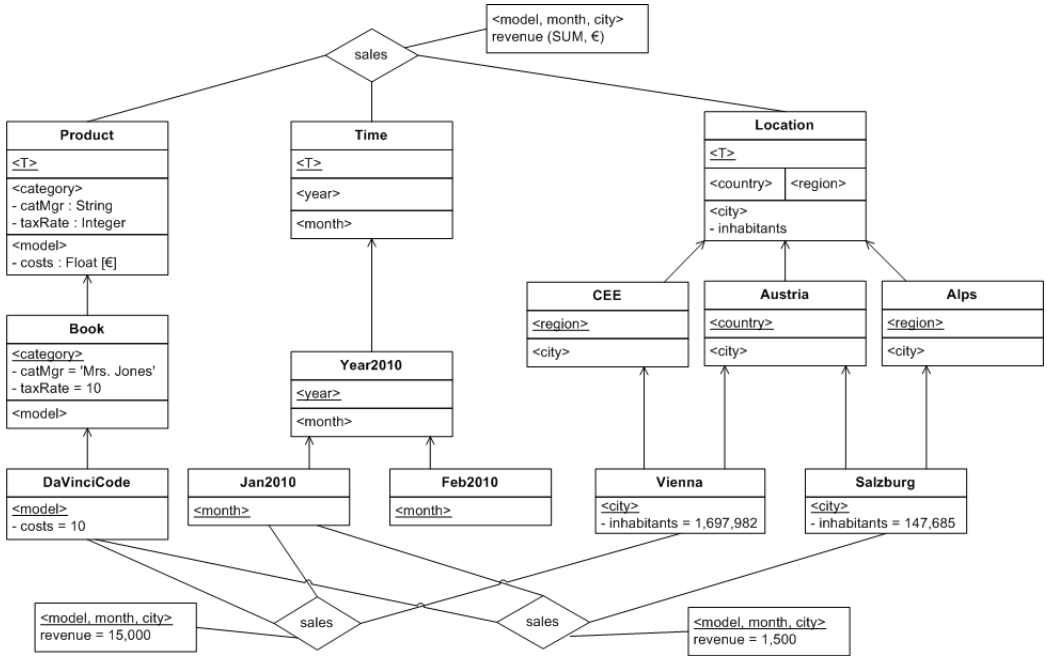


Figure 2.3: Homogeneous three-dimensional *sales* cube (example taken from [NST10])

Consider the example m-relationships as depicted in figure 2.3. The coordinates (*DaVinciCode*, *Jan2010*, *Salzburg*) and (*DaVinciCode*, *Jan2010*, *Vienna*) are descendants of coordinate (*Product*, *Time*, *Location*), written as (*DaVinciCode*, *Jan2010*, *Salzburg*) $\prec$ (*Product*, *Time*, *Location*) and (*DaVinciCode*, *Jan2010*, *Vienna*) $\prec$ (*Product*, *Time*, *Location*). An m-relationship concretizes another m-relationship by replacing some of the coordinate's m-objects with a descendant m-object. For example, the m-relationship connecting m-objects *DaVinciCode*, *Jan2010*, and *Salzburg* replaced m-object *Product* with *DaVinciCode* $\prec$ *Product*, m-object *Time* with *Jan2010* $\prec$ *Time*, and m-object *Location* with *Salzburg* $\prec$ *Location*.

The partial order of connection-levels is given by the level-hierarchies of the m-relationship's dimensions. A connection-level of an n-dimensional m-relationship is an n-tuple of levels. In order for a connection level $\langle l'_1, \ldots, l'_n \rangle \in (L_{o_1} \times \ldots \times L_{o_n})$ to be a descendant of or equal to another connection-level $\langle l_1, \ldots, l_n \rangle \in (L_{o_1} \times \ldots \times L_{o_n})$, written as $\langle l'_1, \ldots, l'_n \rangle \preceq \langle l_1, \ldots, l_n \rangle$, each of the constituting levels of the descendant connection-level in the dimensions $i = 1..n$ has to be a sub-level of or equal to the level of the respective dimension in the parent connection-level, i.e., $\langle l'_1, \ldots, l'_n \rangle \preceq \langle l_1, \ldots, l_n \rangle$, iff $l'_1 \preceq l_1 \wedge \ldots \wedge l'_n \preceq l_n$. The descendant connection-level is said to be of a more specific granularity iff at least one level l'_i is a proper sub-level of l_i, i.e., $l'_i \prec l_i$. A measure m' defined at a connection-level $\langle l'_1, \ldots, l'_n \rangle \prec \langle l_1, \ldots, l_n \rangle$ is thus said to be available at a more specific granularity than measure m at connection-level $\langle l_1, \ldots, l_n \rangle$. For example, connection-level $\langle model, month, city \rangle \in (L_{Product} \times L_{Time} \times L_{Location})$ is a more specific

granularity than connection-level $\langle category, year, country \rangle \in (L_{Product} \times L_{Time} \times L_{Location})$ or even $\langle model, month, country \rangle \in (L_{Product} \times L_{Time} \times L_{Location})$.

Unlike the concretization relationships of m-objects, the concretization relationships between different m-relationships are not stated explicitly. Rather, an m-relationship r' defined at a more specific coordinate than m-relationship r is implicitly said to be a concretization $r' \prec r$. Nevertheless, certain consistency criteria have to be fulfilled by the consistent m-relationship concretization [NST10]:

1. The coordinate of m-relationship r' has to be a proper sub-level of the coordinate of m-relationship r, i.e., $coord(r') \prec coord(r)$, in order for r' to be a consistent concretization of r.

2. The *measure containment* rule specifies that each measure m of an m-relationship $r - m \in M_r$ – is a measure of the concretizing m-relationship r' as well iff its base-level (connection-level) is a sub-level of or equal to the top connection-level of r'. Other measures are not included in the measure set of m-relationship r'. The implementation handles this using inheritance, and the inherited measures do not need to be specified explicitly for the concretizing m-relationship (see chapter 3).

3. The connection-level of a measure m shared by two m-relationships $r' \prec r$ must either be the same or at a more specific granularity at the concretizing m-relationship r' (*assured granularity*). This definition allows a concretizing m-relationship to move a measure to a more specific level of granularity, which is an important ability for representing heterogeneities.

4. Measure types and aggregation functions for a particular measure m must be the same for all m-relationships. Considering the alternative representation using meta-values and default values, for two m-relationships $r' \prec r$ sharing measure $m \in M_r \cap M_{r'}$, where r first introduced m, the measure type is defined at meta level *quantity*, the aggregation function at meta level *function*, the consistency criteria are defined as follows:

   a) $type(u_r(m)) = type(u_{r'}(m)) = mv_r(m, quantity)$ and $\neg def_r(m, quantity)$ (*stability of measure types*)

   b) $f_r(m) = f_{r'}(m) = mv_r(m, function)$ and $\neg def_r(m, function)$ (*stability of aggregation functions*)

2.3.2 Hetero-homogeneous m-cubes

With m-relationships being adapted for the usage in data warehouses, the concept of the multi-level cube (m-cube) can be introduced. The m-cube was first presented in [NST10]. The m-cube $C = (D_1, \ldots, D_n; S_C, R_C)$ consists of n dimensions, a root-coordinate, and a set of m-relationships. The root-coordinate is an n-tuple $(o_1, \ldots, o_n) \in O_{D_1} \times \ldots \times O_{D_n}$. An m-relationship provides a value for a particular measure only if the measure's connection-level is equal to the m-relationship's top-level. Those m-relationships $r \in R_C$ that provide a value for a measure are referred to as *base facts* or *base cells* of the cube C.

The set of *directly subsuming m-relationships* $\hat{R}_r$ of an m-relationship $r \in R_C$ is defined as follows: $\hat{R}_r := \{r' \in R \mid r \preceq r' \wedge \nexists r'' \in R : r \preceq r'' \prec r'\}$.

A homogeneous cube modeled with m-objects is shown in figure 2.3. The root m-relationship connects the root m-objects of each of the cube's dimensions. There are no heterogeneities neither in the aggregation hierarchies of the dimensions nor in the granularity of the measures. Measures are only given at a single connection-level, namely $\langle model, month, city \rangle$. The *sales* cube depicted in 2.3 could also be modeled in the *Dimensional Fact Model* [GMR98] as it presents no heterogeneities.

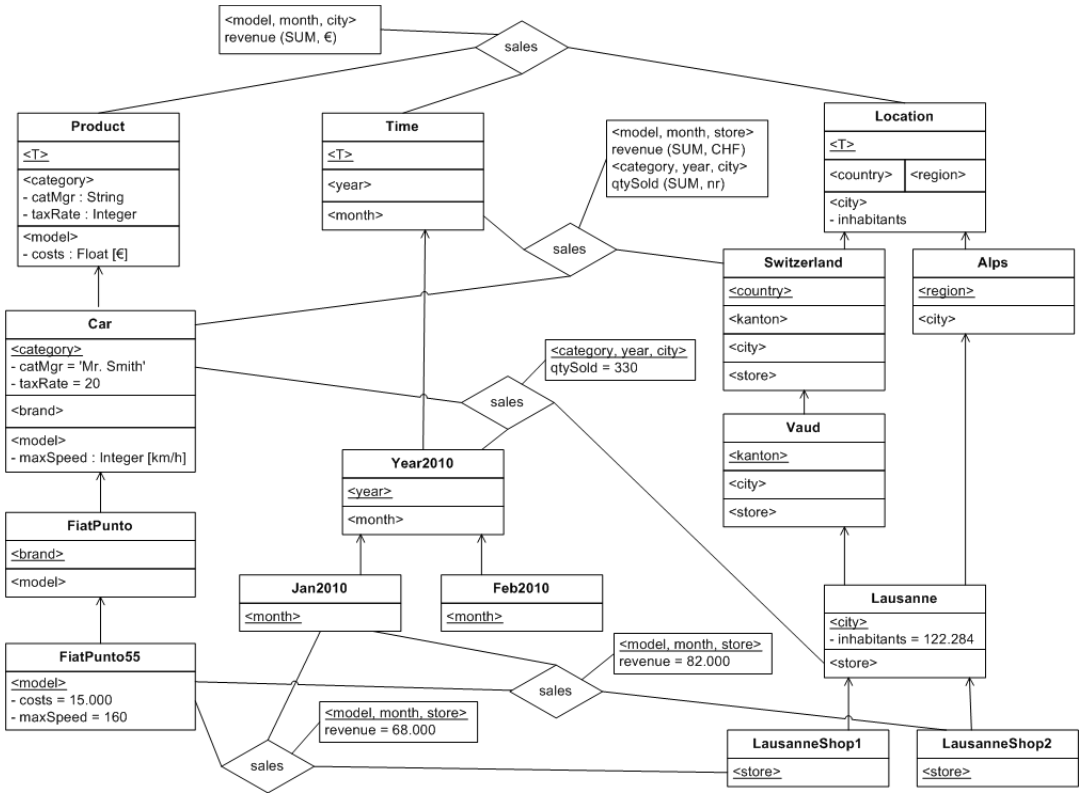


Figure 2.4: Hetero-homogeneous three-dimensional *sales* cube (extended example from [NST10])

Figure 2.4 shows an extended *sales* cube with various heterogeneities. The sub-cube at coordinate (*Car*, *Time*, *Switzerland*) represents car sales in Switzerland. A new m-relationship is introduced at this coordinate in order to define an additional measure *qtySold* and move the existing measure *revenue* to a more specific granularity for this specific sub-cube. Measure *revenue* is moved from connection-level $\langle model, month, city \rangle$ to connection-level $\langle model, month, store \rangle$. Car sales in Switzerland are therefore provided at a higher granularity. A new measure, namely the sold quantity *qtySold*, is introduced for car sales in Switzerland at connection-level $\langle category, year, city \rangle$. This means that the sold quantity of all cars is measured by year and city. It is not necessary for all

measures to be defined at the same connection-level. The unit of measurement *revenue* is furthermore changed to Swiss francs. A hetero-homogeneous m-cube may therefore be heterogeneous with respect to (i) the aggregation hierarchies of the dimensions referenced by the m-cube, (ii) the granularity levels of a particular measure, (iii) the granularity levels of different measures, (iv) the measures that are available at different sub-cubes, and (v) the units of measurement for a particular measure.

Certain consistency criteria are defined for m-cubes in order to avoid overlapping facts and conflicts caused by multiple concretization. M-relationships within the m-cube are unambiguously identified by their coordinate. Two m-relationships are necessarily distinct with respect to their coordinate within the m-cube. The existence of an m-relationship that corresponds to the root-coordinate is obligatory. Each m-relationship of an m-cube has to be defined at a sub-coordinate of the root-coordinate. Note that an m-cube's root-coordinate not necessarily corresponds to the coordinate obtained by combining the root m-objects of the m-cube's dimensions. Measures are introduced at exactly one m-relationship; descendant m-relationships inherit the measure from the ancestors. The following consistency criteria have to be followed by m-cubes in order to be consistent [NST10]:

1. An m-cube C has an m-relationship $r \in R_C$ which is defined at the m-cube's root-coordinate, i.e., $coord(r) = S_C$. This m-relationship may be referred to as root m-relationship.

2. For each coordinate in the m-cube there is at most one m-relationship defined, i.e., $\forall r \in R_C : \nexists r' \in R_C : coord(r) = coord(r')$. The coordinate is the identifier for m-relationships.

3. The m-relationships of the m-cube must follow the rules for the consistent concretization of m-relationships.

4. Measures are introduced at exactly one m-relationship, i.e., two m-relationships sharing a common measure m must have a common ancestor m-relationship that first introduced this measure (*unique induction rule for measures*).

5. Measures shared by two overlapping m-relationships r and r', with $coord(r) \between coord(r')$, must not have a value assigned by both r and r'.

6. For each non-empty cell $x \in X$ – where X is the m-cube's set of coordinates – and a pair of m-relationships from the set of directly subsuming m-relationships of x with these two m-relationships containing a measure m with a base-level under or equal to the connection-level of x, measure unit and base-level have to be the same for both m-relationships, i.e., $\forall x \in X$, $\forall r, r' \in \hat{R}_x$, $\forall m \in M_r \cap M_{r'} : (\exists r \in R : coord(r) \preceq x) \wedge (b_r(m) \preceq \hat{l}_x \vee b_{r'}(m) \preceq \hat{l}_x)) \implies$

   a) $u_r(m) = u_{r'}(m)$ (*unit conflict avoidance*)

   b) $b_r(m) = b_{r'}(m)$ (*base-level conflict avoidance*)

2.4 OLAP with hetero-homogeneous dimension hierarchies and m-cubes

Query operations are provided for extracting information from a hetero-homogeneous data warehouse. A dedicated query algebra has been developed for querying m-cubes. Two types of query operations have to be distinguished. First, query operations for selecting sub-cubes and thus reducing the number of m-relationships and measures are available. Second, query operations for extracting information and calculating aggregate values are provided. The Oracle extension package furthermore includes functionality for exporting m-object dimension hierarchies and m-cubes into relational tables according to the star and snowflake schema, which are not part of the actual query algebra and thus explained in chapter 4.

2.4.1 Branching dimensions

For the implementation of the Oracle extension package the original approach as introduced in [NST10] for employing m-objects to represent cube dimensions has been extended. A new function *dimension branch* is defined that allows to select a sub-branch of an existing dimension. The result is a new consistent dimension containing only m-objects that are descendants of a specified root m-object.

Given a dimension $D = (O_D, A_D, L_D, H_D)$ and an argument m-object $o \in O_D$, function *dimension branch* written as $\beta_o D$ selects a sub-branch of the input dimension D and returns a result dimension $D' = (O_{D'}, H_{D'}, L_{D'}, A_{D'})$. The result dimension D' comprises all descendant m-objects of new root m-object o, given that the top-level is contained within the result dimension's set of levels, i.e., $O_{D'} = \left\{ o' \in O_D \mid o' \prec o \wedge \hat{l}_{o'} \in L_{D'} \right\}$. A level is only included in the result dimension if it is a level of the new root m-object o or if it is not contained in the level set of any m-object that is not a descendant of m-object o, i.e., $L_{D'} = \{l \in L_D \mid \exists o' \in O_D : (o' = o \wedge l \in L_{o'}) \vee \nexists o' \in O_D : (o' \not\prec o \wedge l \in L_{o'})\}$. Attributes are contained in the new dimension D' if their level is contained in the result dimension, i.e., $\{a \in A_D \mid l_D(a) \in L_{D'}\}$ given there is a function $l_D : A_D \to L_D$ that returns the level assigned to an attribute in the dimension D. Given an attribute $a \in A_D$, function l_D is equal to the function l_o of the m-object o that first introduces attribute $a \in A_D$, where $o \in O_D : a \in A_o \wedge \nexists o' \in O_D : (o' \succ o_n \wedge a \in A_{o'})$. The result dimension's m-object hierarchy contains only tuples of m-objects that are in the new dimension's set of m-objects: $H_{D'} = \{(o', o'') \in H_D \mid o' \in O_{D'} \wedge o'' \in O_{D'}\}$.

Some m-objects are disregarded in the result dimension of the *dimension branch* even though the new root m-object is one of their ancestors. This can be the case when levels have multiple parent-levels. Consider, for instance, the *location* dimension where an alternative aggregation path has been defined. Within the *location* dimension, level *city* is a sub-level of both *region* and *country*. On the left-hand side of figure 2.5, the original *location* dimension is depicted. The right-hand side shows the branched *location* dimension after applying function *branch* with m-object *Alps* as the new dimension's root. Notice that attribute *inhabitants* which was originally introduced by m-object *Product*

is introduced by m-object *Alps* in the branch. All m-objects in this *Alps* dimension are descendants of m-object *Alps*. However, not all descendants of m-object *Alps* are included.

In the *location* dimension, m-object *Switzerland* introduces level *store* which is not contained in m-object *Alps*. Including level *store* in the branch dimension would render it inconsistent due to the unique induction rule for levels. M-objects *Lausanne* and *Montreux* would both introduce level *store* independently, which violates the unique induction rule. For the branched dimension in order to be consistent and to contain level *store*, m-object *Alps* would have to define level *store*. However, since m-object *Salzburg* does not define level *store*, the cube would be inconsistent even more so. Thus, level *store* is disregarded and all m-objects whose top-level corresponds to this level are not included in the result dimension.

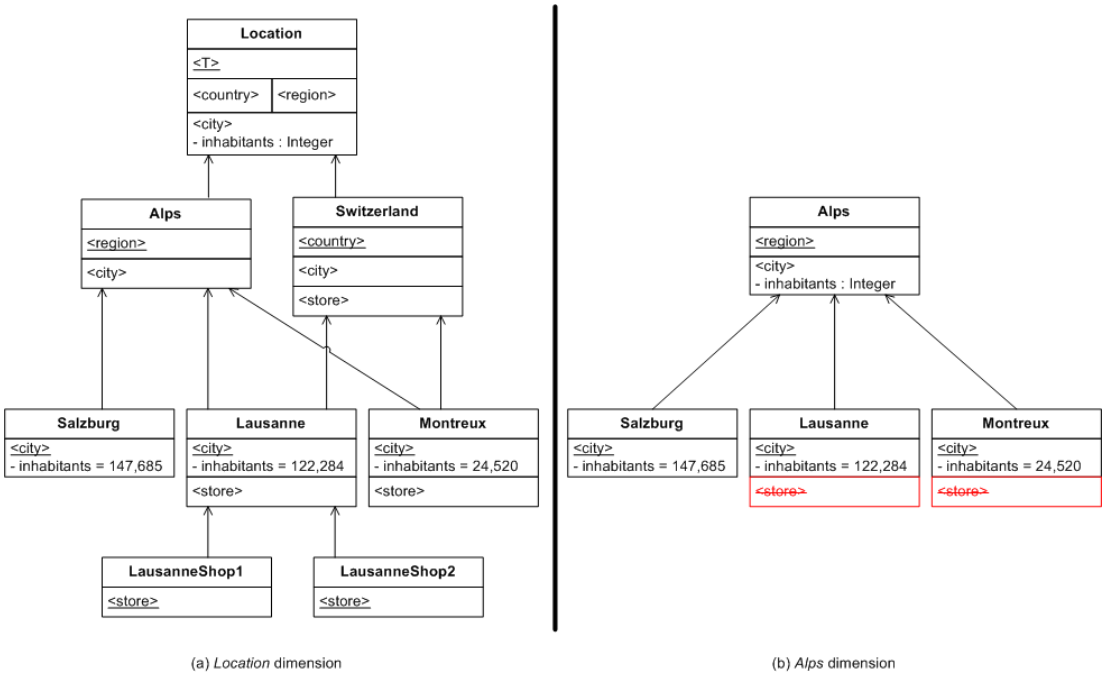


Figure 2.5: The *location* dimension and its branch

2.4.2 Closed m-cube query operations

Closed m-cube query operations are applied on m-cubes in order to obtain a result m-cube with a reduced set of m-relationships and/or measures. The formal definitions of the closed m-cube query operations have been taken from [NST10].

Dice

The *dice* operation $\delta_{o_1,\ldots,o_n}C$ is applied to extract a sub-cube from a given m-cube C with a specified root-coordinate $(o_1,\ldots,o_n)$. Given a m-cube $C=(D_1,\ldots,D_n;S_C,R_C)$ and that there is an m-relationship that corresponds to the argument dice coordinate, operation $\delta_{o_1,\ldots,o_n}C$ returns a result m-cube $C'=(D_1,\ldots,D_n;S_{C'},R_{C'})$, where the result m-cube's root-coordinate $S_{C'}=(o_1,\ldots,o_n)$ and the new m-cube's set of m-relationships $R_{C'}=\{r'\in R_C \mid coord(r')\preceq(o_1,\ldots,o_n)\}$.

For example, $\delta_{Car,Year2010,Switzerland}sales$ selects car sales in Switzerland in the year 2010. The root-coordinate of the result m-cube is (*Car*, *Year2010*, *Switzerland*); all m-relationships at descendant coordinates are included in the result m-cube. The *dice* operation requires the existence of an m-relationship at the argument coordinate. The implementation drops this condition as it transforms the source m-cube in order to obtain a consistent result m-cube. The transformation of the source m-cube's m-relationships is explained in chapter 3.

Projection

The *projection* operation $\pi_{\mathcal{M}}C$ is applied on the source m-cube $C=(D_1,\ldots,D_n;S_C,R_C)$ with an argument set of measures $\mathcal{M}\in M_C$, where $M_C=\bigcup_{r\in R_C}M_r$, and returns a new cube with a reduced set of measures. For each m-relationship in the source m-cube, a corresponding m-relationship in the result m-cube exists including only those measures specified by the query parameter, provided the m-relationship possessed the measure in the first place.

Slice

The selection of m-relationships by the *slice* operation is based on the coordinates of an m-cube. Coordinates are selected based on their compliance with specified selection criteria. All m-relationships defined at coordinates that fulfill the given selection predicates are included in the result m-cube. Furthermore, all descendants and ancestors of the selected coordinates are considered. The *slice* operation may be used to select a 'vertical slice' of an input m-cube with the coordinates satisfying certain criteria. These criteria are boolean expressions over m-object attributes at a specific level. The notions of *upward navigation* and *class extension* are required to be able to formally define the *selection predicates* and the *slice* operation (cf. [NST10]).

Each m-object o represents the class of its (transitive) descendant m-objects at its top-level $\hat{l}_o$. For example, m-object *Car* is the class at level *category* for m-objects *FiatPunto*, *DodgeViper*, *FiatPunto55*, *DodgeViperGTS*, etc. The notion of *upward navigation* $o[l]$ denotes the set of ancestor m-objects of m-object $o\in O_D$ with top-level $l\in L_D$ and is formally defined by the following term: $o[l]:=o':(o,o')\in H_D^*\wedge\hat{l}_{o'}=l$.

The notion of *class extension* represents the set of m-objects with a specified top-level l that are descendants of an m-object o, written as $o\langle l\rangle$. For example, $Book\langle model\rangle$ is the set of descendant m-objects of m-object *Book* with top-level *model*. The class extension is formally defined by the following term: $o\langle l\rangle:=\left\{o' \mid (o',o)\in H_D^*\wedge\hat{l}_{o'}=l\right\}$. A slice

predicate p is a boolean expression over an m-object attribute of a class of m-objects $o\,\langle l \rangle$.

The *slice* operation takes a slice predicate for each of the m-cube's dimensions and assigns a particular level to them, thus obtaining predicate/level pairs, i.e., $(p_1, l_1), \ldots, (p_n, l_n)$. These predicates are used as criteria to select the included coordinates. Slice operation $\sigma_{(p_1,l_1),\ldots,(p_n,l_n)}C$ is applied on the m-cube C using slice predicates $p_1, \ldots, p_n$ at levels $l_1, \ldots, l_n$ to select coordinates of the cube $C = (D_1, \ldots, D_n; S_C, R_C)$ at connection-level $\langle l_1, \ldots, l_n \rangle$ where the constituting m-objects satisfy the specified boolean expressions over their attributes. After obtaining the set of satisfying coordinates at a specific connection-level the set of selected cells is extended with the ancestor and descendant coordinates of the previously selected cells. The result m-cube $C' = (D_1, \ldots, D_n; S_{C'}, R_{C'})$ is returned, with $S_{C'} = S_C$ and the set of m-relationships $R_{C'}$ is comprised of all m-relationships $r \in R_C$ at coordinates that are contained in the set of the selected cells. All cells at the given connection-level satisfying the selection criteria as well as all of their descendant and ancestor coordinates are selected, thus obtaining a 'vertical slice' of the source cube. The selected cells and the included m-relationships are therefore defined as follows [NST10]:

- The selected cells $\bar{X} := \{o \in o_1\,\langle l_1 \mid p_1(o) \rangle\} \times \ldots \times \{o \in o_n\,\langle l_n \mid p_n(o) \rangle\}$.
- The included m-relationships $\bar{R} := \left\{r \in R_C \mid \exists x \in \bar{X} : \mathit{coord}(r) \preceq x\right\}$.
- The set of the result m-cube $R_{C'}$ contains the set of included m-relationships and all ancestor m-relationships: $R_{C'} := \bar{R} \cup \left\{r \in R_C \mid \exists \bar{r} \in \bar{R} : \bar{r} \preceq r\right\}$.

For example, $\sigma_{(\mathit{costs}>70000,\mathit{model}),(\mathit{inhabitants}>100000,\mathit{city})}\mathit{sales}$ may be used to select only sales in big cities of luxury products. Notice that the *slice* operation does not demand a predicate to be specified for all of the input m-cube's dimensions. Considering the previously introduced *sales* cube as depicted in figures 2.3 and 2.4, m-relationships are selected that connect m-objects *DodgeViperGTS*, *DodgeViper*, *Car*, and *Product* of the *product* dimension as well as m-objects *Salzburg*, *Austria*, *Lausanne*, *LausanneShop1*, *LausanneShop2*, *Vaud*, *Switzerland*, *Alps*, and *Location* of the *location* dimension.

2.4.3 Fact extraction and aggregation of measures

Obtaining aggregate values by applying a specific aggregation function on granular data is a common operation in data warehouses. For this purpose, operation *val* is defined [NST10]. This measure aggregation operation is applied on an m-cube $C = (D_1, \ldots, D_n; S_C, R_C)$ and obtains the aggregate value of a given measure m at a given coordinate x. For obtaining the aggregate value, all m-relationships r with $\mathit{coord}(r) \preceq x$ are considered. Aggregation function $f_x(m)$ is used to calculate the aggregated value. Function *val* returns a null value in case that there is no m-relationship r with $\mathit{coord}(r) \preceq x$ where the value $v_r(m)$ is defined. The heterogeneities in the measure units are overcome by converting all measures into a specified target unit u using the convert function

$conv(u_s, u_t, m$. Function *conv* converts a measure value of the source unit u_s into a measure value of the target unit u_t. Function *val* is therefore defined as follows [NST10]:

$$val(m, x, u) := \begin{cases} f_x(m)(\bigcup_{r \in R_x} conv(u_r(m), u, v_r(m))) & \text{if } \exists r \in R_x : (v_r(m) \text{ is defined}) \\ null & \text{otherwise} \end{cases}$$

The original definition of the *measure aggregation* operation allows for a measure unit u to be specified. The granular data is then converted to the specified measure unit using a convert function. This is not yet supported by the prototype. This feature will have to be included in future versions of the extension package, e.g., by extending m-objects with methods in order to provide conversion functions for measure units (see chapter 3). The current implementation allows for the specification of measure units. However, no means for converting between these units is provided by the extension package.

The set of *common measures* at a given coordinate $x = (o_1, \ldots, o_n)$ from an m-cube $C = (D_1, \ldots, D_n; S_C, R_C)$, denoted by M_x, is the union of the set of measures of the *direct subsuming* m-relationships $\hat{R}_x$ of coordinate x. An included measure's connection-level has to be a sub-level of the top connection-level of coordinate x, i.e., $b_r(m) \preceq (\hat{l}_{o_1}, \ldots, \hat{l}_{o_n})$. The set of common measures at coordinate x is then formally defined as follows: $M_x := \left\{ m \in \bigcup_{r \in \hat{R}_x} M_r \mid \forall r \in \hat{R}_x : b_r(m) \preceq (\hat{l}_{o_1}, \ldots, \hat{l}_{o_n}) \right\}$.

The *fact extraction* operation φ is based on the *aggregation of measures* operation. Operation $\varphi_{(o_1,\ldots,o_n)}C$ is applied on an m-cube $C = (D_1, \ldots, D_n; S_C, R_C)$ and returns a relation with one tuple where all measures available at argument coordinate $x = (o_1, \ldots, o_n)$ are rolled-up to coordinate x. A mapping from measures to measure units is given in order to cope with heterogeneities, i.e., $(m_1 \mapsto u_1, \ldots, m_k \mapsto u_k)$ provides a mapping for all measures $m \in M_x$ to measure units. The measures are then converted to the respective measure units before the aggregate value is obtained. The result relation has schema $(D_1, \ldots, D_n, m_1 : u_1, \ldots, m_k : u_k)$ and its instance consists of one tuple $(o_1, \ldots, o_n, val(m_1, x, u_1), \ldots, val(m_k, x, u_k))$. In the original definition, measures are converted to a specific measure unit to account for the heterogeneities in the data warehouse, which is not implemented in the current prototype.

2.4.4 Query views

The implementation of the closed m-cube query operations is primarily done using object-generating query semantics (cf. [SS91]) which demands excessive transformation work to be performed on the m-relationships in order to obtain consistent result m-cubes (see chapter 3). The concept of query views has been conceived in order to provide a more efficient means of querying m-cubes. Since the original definitions of the query operations as given in [NST10] do not consider any implementation aspects, query views are introduced to reflect implementation considerations. A query view is a data structure specifically introduced for the implementation of the m-cube query operations with object-preserving query semantics. The formal definitions of query views can also be found in [Neu10].

An m-cube query view provides the basis for performing object-preserving query operations. A query view $Q = (C, S_Q, R_Q, M_Q)$ holds a reference to an m-cube $C = (D_1, \ldots, D_n; S_C, R_C)$. The query view's root-coordinate $S_Q \in O_{D_1} \times \ldots \times O_{D_n}$ is a sub-coordinate of the referenced m-cube's root-coordinate $S_C \in O_{D_1} \times \ldots \times O_{D_n}$, i.e., $S_Q \preceq S_C$. A query view has a set of references to m-relationships R_Q that is a subset of the referenced m-cube's set of m-relationships R_C, i.e., $R_Q \subseteq R_C$. The query view's set of measures M_Q is a subset of the set of measures M_C given as the union of the measure sets of all of the referenced m-cube's m-relationships.

A query view $Q = (C, S_Q, R_Q, M_Q)$ referencing an m-cube $C = (D_1, \ldots, D_n; S_C, R_C)$ implicitly has a corresponding m-cube $C' = (D_1, \ldots, D_n; S_{C'}, R_{C'})$ defined. The corresponding m-cube C' has the same root-coordinate as the referenced m-cube C, i.e., $S_C = S_{C'}$, and its set of m-relationships $R_{C'} = \{r \in R_C \mid \exists r' \in R_Q : r' \preceq r\}$. The set of measures of each of the included m-relationships $r \in R_{C'}$ only contains measures within M_Q

The same query operations δ *dice*, π *projection*, σ *slice*, and φ *fact extraction* as for m-cubes can be applied on query views. In this context, the closed query operations δ *dice*, π *projection*, and σ *slice* are closed on query views, i.e., they take a query view as input and return a query view as output. The query view operations are refined versions of the original m-cube query algebra introduced in [NST10].

Dice

Applying $\delta_{o_1,\ldots,o_n} Q$ on the query view $Q = (C, S_Q, R_Q, M_Q)$ with an argument coordinate $(o_1, \ldots, o_n) \preceq S_Q$ produces a result query view $Q' = (C, S_{Q'}, R_{Q'}, M_{Q'})$. The m-cube referenced by the result query view Q' is the same m-cube C as referenced by the source view. The result query view's root-coordinate is equal to the argument coordinate, i.e., $S_{Q'} = (o_1, \ldots, o_n)$. The result query view's set of m-relationships contains all m-relationships that are at a sub-coordinate of the argument coordinate, i.e., $R_{Q'} = \{r \in R_Q \mid coord(r) \preceq (o_1, \ldots, o_n)\}$. The set of measures does not change, i.e., $M_{Q'} = M_Q$.

Projection

The *projection* operation $\pi_{\mathcal{M}} Q$ on a query view $Q = (C, S_Q, R_Q, M_Q)$ returns a result query view $Q' = (C, S_{Q'}, R_{Q'}, M_{Q'})$ where the measure set M_Q is equal to the argument set of measures $\mathcal{M}$. The argument set of measures is a sub-set of the referenced m-cube's measure set, i.e., $\mathcal{M} \subseteq M_C$, where the union of the measure sets of all m-relationships of m-cube C, $i = 1..n$, is formed to obtain $M_C = M_{r_1} \cup \ldots \cup M_{r_n}$. The result m-cube's root-coordinate $S_{Q'}$ as well as its set of measures $R_{Q'}$ is unaltered with respect to the source query view.

Slice

The *slice* operation $\sigma_{(p_1,l_1),\ldots,(p_n,l_n)} Q$ on the query view $Q = (C, S_Q, R_Q, M_Q)$ returns a result query view $Q' = (C, S_{Q'}, R_{Q'}, M_{Q'})$ with a reduced set of m-relationships. Only

those m-relationships are contained that are in a concretization relationship with a cell in the set of selected cells. The selected cells are defined as $\bar{X} := \{o \in o_1 \langle l_1 \mid p_1(o)\rangle\} \times \ldots \times \{o \in o_n \langle l_n \mid p_n(o)\rangle\}$ and the included m-relationships are then given as $\bar{R} := \big\{r \in R_Q \mid \exists x \in \bar{X} : \mathit{coord}(r) \preceq x\big\}$. The result query view's set of measures $R_{Q'}$ is defined as follows: $R_{Q'} := \bar{R} \cup \big\{r \in R_Q \mid \exists \bar{r} \in \bar{R} : \bar{r} \preceq r\big\}$. The root-coordinate of the source query view does not change, nor does the set of included measures.

3 A data warehouse extension package for Oracle

Existing database software can be extended in order to support hetero-homogeneous dimension hierarchies and cubes. The feasibility of the concept is explored by conceiving an extension package for the Oracle database software. This chapter introduces the package interface and demonstrates how the user can operate a hetero-homogeneous data warehouse. The basic concept for the implemented extension package was taken from [Neu10].

Operating a hetero-homogeneous data warehouse is a three-step process. First, the dimensional structure of the m-cube is defined by creating dimensions and m-objects. Following the creation of the m-cube and its dimensions and m-objects, the m-relationships are created and their measures introduced. Finally, query operations may be performed on the facts of the m-cube. The prototypical implementation supports all steps of this process. Definition and query operations alike are provided by the extension package.

The extension package's functionality is designed for the use in Oracle PL/SQL programs. The extension package is a collection of PL/SQL packages and object types which enables the user to build data warehouses with hetero-homogeneous hierarchies and m-cubes. Thus, the provided features are intended to be used from within PL/SQL anonymous blocks, procedures or functions. M-cubes, dimensions and m-objects are logically represented by object types. Most of the administration functionality is encapsulated in the member methods of the object types. The usage of the creation, maintenance and query methods is presented in this chapter.

3.1 Defining the dimensional structure of m-cubes

Before being able to create m-cubes and filling them with data, their dimensional structure has to be established. Each m-cube is defined by a number of dimensions. Each dimension in turn is made up of multiple m-objects. Package `mcube` provides the functions to create m-cubes and dimensions (see table 3.1).

The dimensions of an m-cube have to be already defined at the time of the m-cube's creation. References to the dimensions are passed to the m-cube creating function. The number of dimensions stays fixed once the m-cube has been created. While dimensions cannot be added to an m-cube after its creation, m-objects may be added to the dimensions of an m-cube at any time, provided the consistency criteria are met.

Thus, the usual procedure for defining an m-cube is as follows. First, the dimensions are created. Each dimension is identified by a globally unique name, i.e., the name of a dimension has to be unique across multiple m-cubes. Dimensions can be re-used in

different m-cubes. Second, the m-objects that define the aggregation hierarchies are added to the dimensions. Each m-object is identified by a name that is unique within its dimension. Finally, the dimensions are passed to a function that creates the m-cube. Optionally, additional m-objects might be added to the previously defined dimensions.

Method	Parameters	Return	Description
`create_dimension`	name `VARCHAR2`	`dimension_ty`	Create a new dimension.
`create_mcube`	name `VARCHAR2`, dimension references `dimension_trty`	`mcube_ty`	Create a new m-cube using dimension references.
`create_mcube`	name `VARCHAR2`, dimension references `dimension_trty` mobject references `mobject_trty`	`mcube_ty`	Create a new m-cube using dimension references and a root-coordinate (given as `mobject_trty`).
`create_mcube`	name `VARCHAR2`, dimension names `names_tty`	`mcube_ty`	Create a new m-cube using dimension names.
`create_mcube`	name `VARCHAR2`, dimension names `names_tty` mobject names `names_tty`	`mcube_ty`	Create a new m-cube using dimension names and a root-coordinate (given as `names_tty`).
`delete_mcube`	name `VARCHAR2`	–	Delete the specified m-cube.

Table 3.1: Procedures and functions of package `mcube`

3.1.1 Creating dimensions

Object type `dimension_ty` represents dimensions in the Oracle object-relational database. Each dimension is identified by a unique name (attribute `dname`) and provides several methods to maintain the dimension's structure. Furthermore, attribute `levelhierarchy` reflects the dimension's overall level-hierarchy, i.e., the level-hierarchy that takes into account the separate level-hierarchies of all the dimension's m-objects. Dimension names are globally unique in the sense that two different dimensions of different m-cubes may not have equal names. As a result, dimensions can be re-used in different m-cubes. For this purpose, all dimensions are stored in table `dimensions` regardless of the m-cube. Dimension objects can then be retrieved from the `dimensions` table to be used in different m-cubes. The dimensions table is queried using object-relational SQL `SELECT` statements.

The first step in the definition process of a hetero-homogeneous data warehouse is the creation of the dimensions. A dimension is created by invoking the dedicated function of the PL/SQL package `mcube`. Function `create_dimension` in the package `mcube` is used to create new dimensions. The function accepts the dimension's unique name as argument and returns an object of type `dimension_ty`. After its execution, the function will have performed all steps necessary to create and initialize a dimension. A corresponding entry in the `dimensions` table will hold the object which can subsequently be retrieved by executing object-relational SQL statements.

Table type `dimension_trty` is defined as a table of references to dimension objects (`NESTED TABLE OF REF dimension_ty`). Instances of this collection type are used to pass dimension references to methods. Member function `get_reference` of type `dimension_ty` may be used to retrieve the reference to a particular dimension object. Most of the time, this function will be used right after the creation of a dimension object. Alternatively, an SQL `SELECT` statement may be used to retrieve the reference from the `dimensions` table.

Listing 3.1 demonstrates the creation of dimension objects. Function `create_dimension` creates the product dimension for a sales cube that is yet to be created. The created dimension object is inserted automatically into the `dimensions` table. The insertion is transparent to the user. References to the newly created dimension objects are stored within a collection object of nested table type `dimension_trty`. The dimension's member function `get_reference` retrieves the reference to the dimension object.

Listing 3.1: Creating dimensions

```
1  DECLARE
2    dimension     dimension_ty;
3    dimension_ref REF dimension_ty;
4
5    dimensions    dimension_trty := dimension_trty();
6  BEGIN
7    /* create the product dimension */
8    dimension := mcube.create_dimension('product_dim');
9
10   /* get the reference to the dimension */
11   dimension_ref := dimension.get_reference;
12
13   /* add the reference to the list of dimension references */
14   dimensions.EXTEND;
15   dimensions(dimensions.LAST) := dimension_ref;
16
17   /* other dimensions are created analogously */
18   (...)
19 END;
```

3.1.2 Handling m-objects

M-objects are handled using the member methods provided by the object types that represent dimensions and m-objects in the Oracle database. Type `dimension_ty` defines functions and procedures to create, retrieve and delete m-objects. Type `mobject_ty` in turn defines a set of methods to manipulate and extract the information contained within m-objects.

Type `dimension_ty` provides the basic m-object handling functions needed to create and retrieve m-objects. Table 3.2 gives an overview of the functions in type `dimension_ty` that are used to handle a dimension's m-objects. Whereas the functions in `dimension_ty` provide only basic functionality – essentially to create m-objects – type `mobject_ty` and its subtypes define the more advanced methods that are needed to work with m-objects, e.g., handling attributes or defining the order of m-objects.

Function	Parameters	Return	Description
`create_mobject`	name `VARCHAR2`, top-level `VARCHAR2`, parents `mobject_trty`, level-hierarchy `p_tty`	`mobject_ty`	Create a new m-object and insert it into the dimension table.
`get_mobject`	name `VARCHAR2`	`mobject_ty`	Get the m-object with the specified name.
`get_mobject_ref`	name `VARCHAR2`	`REF mobject_ty`	Get a reference to the m-object with the specified name.
`get_root`	–	`mobject_ty`	Get the dimension's root m-object, i.e., an m-object with no parents.
`get_root_ref`	–	`REF mobject_ty`	Get the reference of the dimension's root m-object.
`get_mobjects`	–	`mobject_tty`	Get all m-objects of the dimension.
`get_mobjects_ref`	–	`mobject_tty`	Get references to all the dimension's m-objects.

Table 3.2: M-object handling member functions of `dimension_ty`

Adding m-objects to dimensions

Each dimension defines a table where its constituting m-objects are stored. The table is created at the time of the dimension's creation. These tables are denoted by the term *dimension table* in analogy with dimension tables in traditional (ROLAP) data

warehouses. They are alternatively referred to as a dimension's m-object table in order to avoid confusion with the `dimensions` table that holds all the dimension objects. Both notions are used interchangeably. The m-object tables are generally named after the dimension, e.g., table `product_dim` for the product dimension. The name of a dimension's m-object table is stored within attribute `mobj_table` of type `dimension_ty` and is thus not only given by convention but also explicitly within a table attribute.

M-objects are strictly bound to a particular dimension. The abstract object type `mobject_ty` is used to represent m-objects in the Oracle object-relational database. Each dimension defines its own subtype of this abstract object type. The dimension tables are object tables of the dimension's concretized `mobject_ty`. This enforces type safety. M-objects created for one dimension, i.e., m-objects of a particular subtype of `mobject_ty`, cannot be inserted into another dimension's m-object table. The naming convention for subtypes of `mobject_ty` follows a particular pattern: `mobject_`⟨*name of dimension*⟩`_ty`, e.g., `mobject_product_dim_ty`. This poses a certain limitation on the length of a dimension's name. Since the Oracle databases does not allow entity names that exceed a length of 30 bytes, a dimension's name must be chosen such that the name of its m-object type concretization will not exceed this limit. The maximum length of a dimension's name is actually 20 bytes. The same issue is encountered with m-cube names and will be fixed in future releases.

Dimensions provide a particular function to create m-objects. A number of arguments is required, describing the characteristics of the m-object. First, a (locally) unique identifier must be specified. There must not exist two m-objects with equal names in the same dimension. Two m-objects that belong to distinct dimensions, however, may well have the same name. Second, the m-object's top-level has to be defined. Third, the parent m-objects need to be specified. Finally, the definition of the m-object's level-hierarchy is required. Newly introduced m-objects must comply with the rules for consistent concretization of m-objects and for the consistent dimension as explained in chapter 2. If an m-object violates any consistency rules, a self-defined Oracle error code (*ORA-code*) with a custom error message will be thrown.

M-objects are created using the `create_mobject` function of `dimension_ty`. This function creates an object of type `mobject_ty`, inserts it into the dimension's m-objects table and returns the newly created object. Using the dimension's `create_mobject` function is equal to invoking the constructor function of the dimension's concretized m-object type (`mobject_*_ty`) and subsequently inserting the object with an `INSERT` statement into the dimension table. However, in the latter case the user has to take care of creating an m-object of the correct type. It is thus encouraged - and more convenient - to use the dimension's create function.

An m-object's level-hierarchy is described by dedicated object types. Type `p_ty` represents a parent-child relationship between two levels of an m-object. The first attribute (`lvl`) of type `VARCHAR2` describes one of the m-object's levels, the other attribute (`parentlevel`) defines the level's parent level. For an m-object's top-level, the parent-level is not contained within the m-object's own level-hierarchy as a value of attribute `lvl`. If a particular level has no parent level, attribute `parentlevel` is `NULL`. Table type `p_tty` of object type `p_ty` represents the actual level-hierarchy. The m-object type's con-

Listing 3.2: Creating the root m-object in the product dimension

```
1  DECLARE
2    dimension dimension_ty;
3
4    mobject mobject_ty;
5
6    levelhierarchy p_tty;
7  BEGIN
8    /* get the dimension object from the dimensions table */
9    SELECT VALUE(dim) INTO dimension
10   FROM   dimensions dim
11   WHERE  dim.dname = 'product_dim';
12
13   /* define the new m-object's level-hierarchy */
14   levelhierarchy := p_tty(p_ty('top', NULL),
15                           p_ty('category', 'top'),
16                           p_ty('model', 'category'));
17
18   /* create the dimension's root object */
19   mobject := dimension.create_mobject('Product',
20                                       'top',
21                                       NULL,
22                                       levelhierarchy);
23 END;
```

venience function `has_level` may be used to check whether an m-object's level-hierarchy contains a specified level. It takes the name of the level as argument and returns `TRUE` if the level exists within the m-object's level-hierarchy; otherwise, `FALSE` is returned. Function `introduced_level` checks whether a particular level is not only contained within the m-object's level-hierarchy but also if the m-object is the highest up in the hierarchy that defines it, i.e., introduced the level. Line 14 in listing 3.2 illustrates how the level-hierarchy for m-object *Product* is defined.

An m-object may introduce new levels with respect to its parent m-objects. In listing 3.2, a level-hierarchy is defined for root m-object *Product* consisting of three levels: *top, category, model.* M-object *Car* now introduces a new level: *brand.* The procedure is shown in listing 3.3 line 18. The child m-object's level-hierarchy specifies the parent-child relationships between the levels, starting with the m-object's top-level. The first parent-child relationship in m-object *Car*'s level-hierarchy consists of the m-object's top-level – *category* – and its parent level – *top* – which is defined in the parent m-object *Product.* Instead of being a (direct) sublevel to level *category*, level *model* of m-object *Car* has level *brand* as parent level in the new level-hierarchy.

The level-hierarchy of a dimension is calculated from the level hierarchies of its constituting m-objects. Thus, every time a new m-object is added to the dimension, its level-hierarchy has to be re-assessed. For instance, after m-object *Product* was added to the product dimension in listing 3.2, the dimension's level-hierarchy consists of the three levels introduced by m-object *Product.* After the creation of m-object *Car* which introduces a new level *brand* the dimension's level-hierarchy will consist of four levels: *top, category, brand, model.* A dimension's level-hierarchy is updated automatically every time an m-object is added to the dimension table; a trigger is used to accomplish this. The update of the dimension's level-hierarchy in case of an m-object's deletion is currently not supported by the prototype. Object type `dimension_ty` stores the level-hierarchy within attribute `levelhierarchy` of type `p_tty`.

Parent m-objects are specified as a list of references to m-objects. Table type `mobject_trty` is provided for that purpose. The root m-object of a dimension, i.e., an m-object with no parent m-objects, takes `NULL` or an empty collection. Line 19 in listing 3.2 demonstrates how the root m-object of the product dimension is created with no parents. Line 18 in listing 3.3 has a collection of type `mobject_trty` being filled with the reference to the previously created root m-object. The reference to an m-object of a particular dimension can be retrieved by calling the dimension's `get_mobject_ref` function, taking the m-object's name as argument. Function `get_mobject` takes the desired m-object's name and returns a copy of the object from the dimension's m-objects table. Function `get_root` of object type `dimension_ty` retrieves a dimension's root m-object; function `get_root_ref` retrieves the object reference.

Levels may also have multiple parent levels. This can be used to model alternative aggregation paths. For instance, consider a *location* dimension. M-object *Location* is the dimension's root m-object and defines levels *top*, *country*, *region*, and *city.* The particularity is that level *city* is a sub-level of both level *country* and level *region*, while levels *country* and *region* are in no hierarchical order to each other. Listing 3.4 illustrates how multiple parent levels for m-objects can be defined. The level-hierarchy contains

Listing 3.3: Creating m-objects in the product dimension

```
1  DECLARE
2    dimension dimension_ty;
3
4    mobject mobject_ty;
5    parents mobject_trty;
6
7    levelhierarchy p_tty;
8  BEGIN
9    (...)
10
11   /* define parents for m-object 'Car'*/
12   parents := mobject_trty();
13   parents.extend;
14   parents(parents.LAST) :=
15     dimension.get_mobject_ref('Product');
16
17   /* define the level-hierarchy for m-object 'Car' */
18   levelhierarchy := p_tty(p_ty('category', 'top'),
19                           p_ty('brand', 'category'),
20                           p_ty('model', 'brand'));
21
22   /* create m-object 'Car' in product dimension */
23   mobject := dimension.create_mobject('Car',
24                                       'category',
25                                       parents,
26                                       levelhierarchy);
27   (...)
28 END;
```

two tuples where attribute `lvl` of type `p_ty` in nested table `levelhierarchy` equals *city*. The first entry defines level *country*, the second entry defines level *region* as level *city*'s parent level. Levels *country* and *region* are declared as sub-levels of *top* in the m-object's level-hierarchy. M-objects with top-level *city* have to reference two parents: a *region* and a *country*. Listing 3.5 demonstrates how m-object *Salzburg* with top-level *city* is created. In contrast to the previous examples involving the creation of m-objects, the list of parents (`mobjects_trty`) of the m-object contains two entries: the references two m-objects *Austria* and *Alps*.

Listing 3.4: Creating the root m-object of the *location* dimension

```
1  DECLARE
2    dimension dimension_ty;
3
4    mobject mobject_ty;
5
6    levelhierarchy p_tty;
7  BEGIN
8    /* get the dimension object from the dimensions table */
9    SELECT VALUE(dim) INTO dimension
10   FROM   dimensions dim
11   WHERE  dim.dname = 'location_dim';
12
13   /* define the new m-object's level-hierarchy,
14      notice that city has two entries */
15   levelhierarchy := p_tty(p_ty('top', NULL),
16                           p_ty('country', 'top'),
17                           p_ty('region', 'top'),
18                           p_ty('city', 'country'),
19                           p_ty('city', 'region'));
20
21   /* create the dimension's root object */
22   mobject := dimension.create_mobject('Location',
23                                       'top',
24                                       NULL,
25                                       levelhierarchy);
26 END;
```

Each m-object knows the dimension it belongs to. A reference to the m-object's dimension (attribute `dim`) is stored by each m-object. The `persist` procedure can be used to persist changes made to the m-object. Alternatively, an SQL `UPDATE` statement may be used. Procedure `delete_mobject` removes the m-object from the dimension table and drops all attribute tables. The dimension must not be changed. Changing the dimension will make the m-object and its dimension unusable.

Listing 3.5: Creating m-objects in the location dimension

```
1  DECLARE
2    dimension dimension_ty;
3
4    mobject mobject_ty;
5    parents mobject_trty := mobject_trty();
6
7    levelhierarchy p_tty;
8  BEGIN
9    (...)
10
11   /* define parents for m-object 'Salzburg',
12      notice that two parent m-objects are given */
13   parents.extend;
14   parents(parents.LAST) :=
15     dimension.get_mobject_ref('Austria');
16
17   parents.extend;
18   parents(parents.LAST) :=
19     dimension.get_mobject_ref('Alps');
20
21   /* define the level-hierarchy for m-object 'Salzburg' */
22   levelhierarchy := p_tty(p_ty('city', 'country'),
23                           p_ty('city', 'region'));
24
25   /* create m-object 'Salzburg' in location dimension */
26   mobject := dimension.create_mobject('Salzburg',
27                                       'city',
28                                       parents,
29                                       levelhierarchy);
30   (...)
31 END;
```

M-objects can be ordered according to the order of their top-levels in the dimension's overall level-hierarchy. Order member function `compare_to` is used to determine the order of m-objects. If the first m-object's top-level is a (transitive) parent level – i.e., ancestor level – of the other m-object's top-level within the dimension's overall level-hierarchy, then the first m-object comes before the other m-object and the `compare_to` function returns a value of -1. On the other hand, the first m-object comes after the other m-object if the first m-object's top-level is a (transitive) sublevel of the other m-object's top-level. In this case function `compare_to` returns a value of 1. In any other case, two m-objects are of equal order. In this case function `compare_to` returns 0. Only m-objects of the same dimension can be compared reasonably. Ordering m-objects in SQL statements through the specification of the `ORDER BY` clause is supported. Given two m-objects o and o' of the same dimension, function `compare_to` is therefore defined as follows:

$$o.compare\_to(o') := \begin{cases} -1 & \text{iff } o.toplevel \prec o'.toplevel \\ 1 & \text{iff } o.toplevel \succ o'.toplevel \\ 0 & \text{otherwise} \end{cases}$$

Handling attributes of m-objects

Attributes of m-objects carry additional information about the level of granularity represented by the m-object. For instance, at the *category* level of granularity, each product has a tax rate. M-object attributes cannot be used for aggregation. They are equivalent to *non-dimension attributes* in the *Dimensional Fact Model* [GMR98].

Attributes are created using the `add_attribute` procedure provided by the m-object. Table 3.3 lists the methods provided by object type `mobject_ty` to handle attributes. The creation of an attribute requires the user to specify a unique name, the level it is defined for, and the data type. An m-object can define attributes for any level within its level-hierarchy, i.e., for any level where a tuple in the m-object's nested table `levelhierarchy` exists where the level is contained in column `lvl`. Attributes must adhere to the consistency criteria as explained in chapter 2. Note that attributes are inherited by concretizing m-objects and must not be added to the descendant m-object again; doing so will result in an error.

The value of an attribute can be set for a particular m-object by invoking the m-object's `set_attribute` procedure. The procedure is overloaded. In its basic variation, the procedure takes as arguments the attribute's name and the value to be set. The alternative signature of this procedure is used to set metadata for the attributes. Certain conditions have to be satisfied for an attribute in order to be assigned a value for a particular m-object. First, the attribute must have been added either to the m-object for which the value is to be set or one of its parent m-objects, i.e., an inherited attribute. Second, the attribute must have been defined for the m-object's top-level – unless the value to be set is metadata.

Consider the product dimension that has been defined in the previous examples. Listing 3.6 demonstrates how attribute *costs* is introduced at level *model* by m-object *Prod-*

Method	Parameters	Return	Description
`add_attribute`	name `VARCHAR2`, level `VARCHAR2`, data type `VARCHAR2`	–	Introduce a new attribute at the specified level.
`set_attribute`	name `VARCHAR2`, value `ANYDATA`	–	Set the value for an added attribute.
`set_attribute`	name `VARCHAR2`, meta level `VARCHAR2`, default? `BOOLEAN` value `ANYDATA`	–	Set metadata or the value for an added attribute.
`get_attribute`	name `VARCHAR2`	`ANYDATA`	Get the value for the specified attribute.
`delete_attribute`	name `VARCHAR2`	–	Remove the specified attribute from the m-object.
`has_attribute`	name `VARCHAR2`, toplevel? `BOOLEAN`, introduced? `BOOLEAN` description `OUT attribute_ty`	`BOOLEAN`	Look if the m-object introduces or inherits an attribute and output its description.
`has_attribute`	name `VARCHAR2`, toplevel? `INTEGER`, introduced? `INTEGER`	`INTEGER`	SQL-compatible version of `has_attribute`.
`list_attributes`	toplevel? `BOOLEAN`, introduced? `BOOLEAN`	`attribute_tty`	List introduced or inherited attributes.

Table 3.3: Attribute handling member methods of `mobject_ty`

uct. Thus, all m-objects that are descendants of *Product* and have top-level *model* may provide an asserted value for attribute *costs*. Attribute *costs* is inherited by the descendant m-objects which must not add this attribute again. Similarly, attribute *taxRate* is introduced at the *category* level. In listing 3.7 line 8, car model *FiatPunto55* with top-level *model* sets a value for attribute *costs*. The attempt in listing 3.7 line 21 to set a value for attribute *costs* for m-object *Car* will raise an error since this m-object's top-level is *category*. However, m-object *Car* has to provide a value for attribute *taxRate* which has been defined by one of its ancestors – m-object *Product* – at level *category* – the m-object's top-level.

M-objects furthermore provide functions to read attribute values and to remove attributes. Function `get_attribute` reads the value of a particular attribute. It is the preferred way to retrieve attribute values. The function returns data of generic type `ANYDATA`. Procedure `delete_attribute` on the other hand removes an attribute from an m-object. All values and metadata set by the m-object or one of its subtypes will be

Listing 3.6: Adding attributes to m-object *Product*

```
1  BEGIN
2    (...)
3
4    /* get m-object 'Product' */
5    mobject := dimension.get_mobject('Product');
6
7    /* products set a value for attribute 'costs'
8       at the model level */
9    mobject.add_attribute('costs', 'model', 'NUMBER(9,2)');
10
11   /* products all have a tax rate
12      at the category level */
13   mobject.add_attribute('taxRate',
14                         'category',
15                         'NUMBER(9,2)');
16 END;
```

deleted as well. Only attributes introduced by the m-object whose `delete_attribute` procedure is called can be deleted.

Function `has_attribute` can be used to find out if a certain m-object introduces or inherits a particular attribute. The query can be further restricted such that only attributes that are defined for the m-object's top-level are considered. Likewise, the query can be restricted to consider only attributes defined by the m-object itself and thus disregards attributes defined by ancestor m-objects. The `has_attribute` function returns a description of the attribute of type `attribute_ty`.

Function `list_attributes` is similar in purpose to the `has_attribute` function. The `list_attributes` function lists all introduced or inherited attributes of a particular m-object. The query can be restricted such that only attributes are listed that are defined for the m-object's top-level and/or only attributes introduced by the m-object itself, i.e., no inherited attributes. Nested table `attribute_tty` of object type `attribute_ty` is returned as result, containing some information about each of the attributes found by the function.

Object type `attribute_ty` is used to describe attributes of m-objects. Various information about an m-object's attribute is stored in type `attribute_ty`. First, the attribute's name (field `attr_name`) and level (field `attr_lvl`) are stored. Second, `attribute_ty` contains the name of the table where the values for this attribute are stored (field `lvltable`). Furthermore, three fields of `attribute_ty` contain information about the attribute's data type. Field `data_type` defines the data type, e.g., `NUMBER` or `VARCHAR2`. Field `data_length` defines the attribute's length, i.e., the number of digits or the number of characters, respectively. Field `data_scale` contains information about the attribute's scale, i.e., the number of digits on the right side of the decimal point.

Listing 3.7: Setting m-object attribute values

```
1  BEGIN
2    (...)
3
4    /* get m-object 'FiatPunto55' at model level */
5    mobject := dimension.get_mobject('FiatPunto55');
6
7    /* provide a value for attribute 'costs' */
8    mobject.set_attribute('costs',
9                          ANYDATA.convertNumber(15000));
10
11   /* get m-object 'Car' at category level */
12   mobject := dimension.get_mobject('Car');
13
14   /* m-object 'Car' gives a value for attribute
15      'taxRate' which is defined at level 'category' */
16   mobject.set_attribute('taxRate',
17                         ANYDATA.convertNumber(20));
18
19   /* the following command, however, causes an error
20      since the top-level of m-object 'Car' is 'category' */
21   mobject.set_attribute('costs',
22                         ANYDATA.convertNumber(15000));
23 END;
```

Attributes of m-objects are not stored in the dimension tables. Rather, attributes are stored in dedicated attribute tables. Consequently, reading and manipulating attribute data using SQL statements requires some knowledge about how the extension stores attribute values. It is therefore recommended to stick to the functions provided by the m-object itself. The logical organization of m-object data is discussed in chapter 4, section 4.2.

Handling metadata about attributes

Metadata can be set in order to provide more detailed knowledge about an attribute. Metadata may be referred to as *attributes to attributes*. For instance, attribute *costs* may be described by metadata € at meta level *unit of measurement*, which states that costs are measured in euros. Furthermore, attribute *costs* may be described by metadata *currency* at meta meta level *quantity*.

Metadata are either default or shared. Default values may be changed by m-objects at a more specific level. Shared values cannot be changed by concretized m-objects. Shared values thus are the opposite of default values. They represent the final values of measures.

An alternative signature of procedure `set_attribute` is used to set attribute metadata. The alternative signature of procedure `set_attribute` requires the user to specify the attribute's name, a meta level (`VARCHAR2`), whether the value given shall be regarded as a default or as a shared value (`BOOLEAN`), and the value itself (`ANYDATA`). A meta level `NULL` means that it is not metadata which are being set by the procedure. Calling the four-parameter signature of procedure `set_attribute` with meta level `NULL` and default value set to `FALSE` is equal to calling the two-parameter signature of the procedure. For instance, issuing the call `set_attribute('costs', NULL, FALSE, ANYDATA.convertNumber(15000))` sets the costs for the m-object it is called upon to 15000. It is equal to the procedure call presented in listing 3.7 line 8.

Metadata, unlike ordinary attribute values, can be set for attributes at any level of an m-object – not only the m-object's top-level. Of course, the attribute whose values are to be set need to be inherited by or defined at the m-object for whom the values shall be set. Moreover, metadata set by one m-object may be changed by one or more of its descendant m-objects. For instance, whereas m-object *Product* defines value € at meta level *unit of measurement* for attribute *costs*, m-objects at a more specific level might define value *$* at the same meta level for the same attribute. However, this modification is only possible if the entry in the metadata nested table has been marked default by m-object *Product*. Otherwise, an error message will be thrown.

Listings 3.8 and 3.9 illustrate how attribute metadata and default values may be set using the provided procedures. The declaration parts of the PL/SQL blocks are straight forward and omitted due to space considerations. Listing 3.8 illustrates how metadata for the same attribute can be set at different meta levels. Metadata for attribute *costs* of m-object *Product* are given. The costs of a product are generally measured in euros. As the default argument is set, ancestor m-objects may override this value, which is done by m-object *DodgeViper* at the brand level in listing 3.9. The costs of all cars of

brand *DodgeViper* are consequently measured in U.S. dollars. This, however, cannot be changed by m-objects at more specific levels since it has been marked shared (or not default).

Listing 3.8: Setting m-object attribute metadata

```
1  BEGIN
2    (...)
3
4    /* get m-object 'Product' at top-level */
5    mobject := dimension.get_mobject('Product');
6
7    /* costs for products are generally measured in euros.
8       however, more specific m-objects may change this */
9    mobject.set_attribute('costs',
10                         'unit',
11                         TRUE,  /* mark as default value */
12                         ANYDATA.convertVarchar2('€'));
13
14   /* meta meta value for 'costs' is 'currency'.
15      this may not be changed by child m-objects */
16   mobject.set_attribute('costs',
17                         'quantity',
18                         FALSE, /* mark as shared value */
19                         ANYDATA.convertVarchar2('currency'));
20 END;
```

Using m-objects to represent attribute metadata

Metadata represent hierarchical information and can therefore be represented by an m-object hierarchy. The use of m-objects for further describing an m-object's attributes is supported since the `attribute_metadata` nested table stores values as `ANYDATA`. In future releases, this feature is scheduled to take over the task completely. The advantage of using m-objects to represent attribute metadata is the reliance on m-object consistency check mechanisms and the possibility to more comprehensively define metadata on metadata.

A *metadata hierarchy* is recommended to be used for representing attribute metadata (figure 3.1). The root m-object of that hierarchy should be a *generic* m-object, defining only a top-level and level *metadata*. Level *metadata* is subsequently used to define different sub-hierarchies of metadata, e.g., an m-object *Measurement* that defines a measurement hierarchy. The descendant m-objects of m-object *Metadata* define the actual metadata level-hierarchy. For instance, a measure system consists of the quantities that are to be measured (level *quantity*) and the units used to measure them (level *unit*).

Listing 3.9: Overwriting default values

```
1  BEGIN
2    (...)
3
4    /* get m-object 'DodgeViper' at top-level brand */
5    mobject := dimension.get_mobject('DodgeViper');
6
7    /* costs for cars of this brand are measured in
8       dollars. models may not further change this. */
9    mobject.set_attribute('costs',
10                         'unit',
11                         FALSE,
12                         ANYDATA.convertVarchar2('$'));
13 END;
```

Figure 3.1 illustrates how measure units and measured quantities used in the sales cube would be grouped within the proposed metadata hierarchy.

M-objects can be set as attribute metadata using the common `set_attribute` overloaded version provided for setting metadata. Argument `metalevel` in this case describes the top-level of the m-object, argument `attr_value` is a reference to the m-object. For instance, the code snippet in listing 3.10 illustrates how function `set_attribute` sets the unit of attribute *costs* at m-object *Product* to be euro by default using m-objects. Suppose that variable `ref_euro` is of type `REF mobject_ty` and that the metadata m-object hierarchy is stored in table `metadata_hierarchy`.

In the current prototypical implementation, the usage of m-objects for representing metadata hierarchies is not enforced. It is, however, very likely that future versions will completely rely on such a system since this approach provides many advantages besides the ability to use m-object consistency check mechanisms. Foremost, future versions of the implementation will support the definition of functions and procedures for m-objects by the user. In this case, functions could be defined for m-objects that take over the task of converting units of measurement. For instance, a function might be defined that converts U.S. dollars into euros and vice versa. This feature is important for performing OLAP operations on heterogeneous data for it is the only means to provide accurate analysis data.

3.1.3 Creating m-cubes

M-cubes can be created as soon as all the dimension objects that constitute the cube have been created. Similar to the creation of dimensions, package `mcube` provides a function for creating m-cube objects which transparently handles all the initialization work in the background.

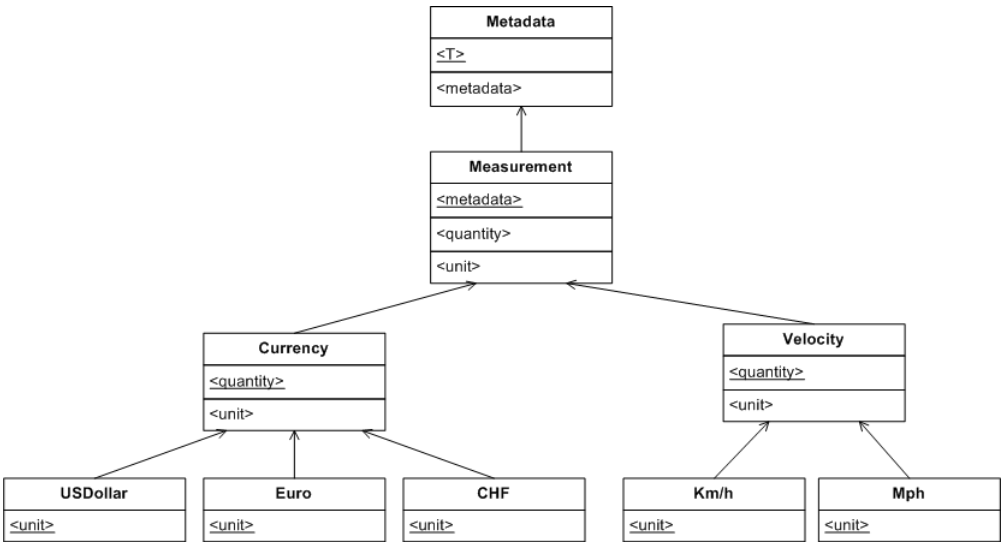


Figure 3.1: Proposed structure of the *metadata hierarchy*

Listing 3.10: Setting attribute metadata using an m-object hierarchy

```
1  DECLARE
2    ref_euro  REF mobject_ty;
3    mobj_prod mobject_ty;
4  BEGIN
5    (...)
6
7    SELECT REF(o) INTO ref_euro
8    FROM   metadata_hierarchy
9    WHERE  o.oname = 'Euro';
10
11   mobj_prod.set_attribute('costs',
12                           'unit',
13                           TRUE,
14                           ANYDATA.convertRef(ref_euro));
15 END;
```

Listing 3.11: Creating m-cubes

```
1  DECLARE
2    product_dim   dimension_ty;
3    time_dim      dimension_ty;
4    location_dim  dimension_ty;
5
6    dimensions    dimension_trty;
7    root_mobjects mobject_trty;
8
9    mc mcube_ty;
10 BEGIN
11   /* dimension references are already obtained */
12   /* product_dim contains dimension product_dim */
13   (...)
14
15   /* root of product dimension in cube is 'Car' */
16   root_mobjects.EXTEND;
17   root_mobjects(root_mobjects.LAST) :=
18       product_dim.get_mobject_ref('Car');
19
20   /* roots of other dimensions in the cube
21      are the the dimensions' actual roots */
22   root_mobjects.EXTEND;
23   root_mobjects(root_mobjects.LAST) :=
24       time_dim.get_root_ref;
25
26   root_mobjects.EXTEND;
27   root_mobjects(root_mobjects.LAST) :=
28       location_dim.get_root_ref;
29
30   /* create a sales cube with explicit root coordinate */
31   mc := mcube.create_mcube('car_sales',
32                            dimensions,
33                            root_mobjects);
34
35   /* create a sales cube with implicit root coordinate */
36   mc := mcube.create_mcube('sales_cube', dimensions);
37 END;
```

Function `create_mcube` in package mcube is used to create m-cube objects. It expects the cube's unique name as argument and either takes a list of references to dimensions (`dimension_trty`) or a list of dimension names. The former is shown in listing 3.11 line 31, where a number of dimensions is passed to function `create_mcube`. The creation of dimensions is illustrated in previous examples and it is assumed that the references to the dimensions are already obtained. A dedicated nested table type has been defined to pass the latter: Type `names_tty` is a table of strings (`VARCHAR2`). In this case function `create_mcube` assumes the task of retrieving the references to the specified dimensions. Furthermore, an m-cube's root-coordinate may be specified as a third parameter.

The abstract object type `mcube_ty` defines the generic structure of m-cubes in the Oracle object-relational database. Each m-cube is identified by a unique name (attribute `cname`). Since each m-cube has different dimensions, function `mcube_ty` creates a new concretization of the `mcube_ty` base type every time an m-cube is defined. The attributes, functions and procedures are concretized with respect to the dimensions passed to the `create_mcube` function. Each of the m-cube's dimensions is referenced by a separate attribute of the `mcube_ty` concretization. These attributes bear the same name as the dimensions they reference, e.g., attribute `product_dim` stores a reference to the object representing the product dimension (see listing 3.12). Likewise, the methods for creating and maintaining m-relationships are concretized with respect to the dimensions of the m-cube. The naming scheme for the `mcube_ty` subtypes is as follows: `mcube_`⟨*name of m-cube*⟩`_ty`, e.g., `mcube_sales_cube_ty`. The user has to be aware of the fact that the naming scheme of the dynamically created m-cube objects poses a constraint on the length of an m-cube's name for the Oracle database imposes a limit of 30 bytes on database entity names. Since m-relationship types and their related object types are as well created dynamically, bearing the m-cube's name within their own name, the maximum length of an m-cube's name is further lowered. The maximum length of an m-cube's name is eventually 12 bytes. This issue will be fixed in future releases.

The number of dimensions of an m-cube stays fixed once created and cannot be altered. The dimension objects referenced by an m-cube must not be changed after creation. Doing so will result in the m-cube being unusable, i.e., the user will not be able to add m-relationships that take into account the new dimensional structure.

An m-cube always has a root-coordinate. The root-coordinate describes the most universal level of granularity within an m-cube. All m-relationships defined within the m-cube are at a sub-coordinate of the root-coordinate. An m-cube's root-coordinate is of type `coord_*_ty` (see section 3.2) and is stored in attribute `root_coord` in `mcube_*_ty`. Function `create_mcube` of package `mcube` provides an alternative signature that lets the user specify a root-coordinate. Since type `coord_*_ty` is a dynamic concretization and cannot be used before the cube is created, it cannot be used to pass an m-cube's root-coordinate to function `create_mcube`. Thus, the additional parameter is a list of m-object references (`mobject_trty`) or alternatively a list of m-object names that constitute the new m-cube's root-coordinate. The entries of this list have to be in the same order as the entries in the parameter list containing the dimension references (`dimension_trty`. If the optional parameter is omitted or set to `NULL`, the m-cube implicitly assumes a root-coordinate with m-objects that correspond to the root m-objects of the m-cube's

dimensions. In this case, attribute `root_coord` of an m-cube is `NULL` and the m-cube internally uses a coordinate that consists of the dimensions' root m-objects whenever a root-coordinate is required. M-cubes that do not define a root-coordinate can be created before the actual m-objects are created. Listing 3.1 illustrates how an m-cube is created both with (line 31) and without (line 36) an explicit root-coordinate.

Function `create_mcube` keeps all initialization work transparent to the user. A subtype of object type `mcube_ty` is generated, an m-cube object of this concretized object type created and the new instance returned as the result of the function `create_mcube`. Each m-cube object is stored in the `mcubes` table and can be retrieved at any time issuing an SQL `SELECT` statement.

Listing 3.12 illustrates how m-cubes can be retrieved from table `mcubes` and how the concretizations of `mcube_ty` may be used to retrieve dimensions. In line 11 the sales cube object is retrieved from the `mcubes` table by issuing a `SELECT` statement. Note that a type cast is necessary since the `mcubes` table is a table of the abstract super type `mcube_ty`. Line 20 demonstrates how a reference to one of the m-cube's dimensions can be accessed and how the reference is used to retrieve the corresponding dimension object.

Listing 3.12: Getting the sales m-cube and its product dimension

```
1  DECLARE
2    sales_cube mcube_sales_cube_ty;
3
4    dimension_ref REF dimension_ty;
5    dimension dimension_ty;
6  BEGIN
7    /* select the sales cube from the table.
8       NOTE: type cast is needed as mcubes is
9             a table of mcube_ty */
10
11   SELECT TREAT(VALUE(mc) AS mcube_sales_cube_ty)
12   INTO   sales_cube
13   FROM   mcubes mc
14   WHERE  mc.cname = 'sales_cube';
15
16
17   /* use the reference attribute in mcube_sales_cube_ty
18      to get the product dimension object */
19
20   dimension_ref := sales_cube.product_dim;
21
22   utl_ref.select_object(dimension_ref,
23                         dimension);
24 END;
```

3.2 Defining facts and measures in m-cubes

Facts in hetero-homogeneous m-cubes are represented by using the concept of *m-relationships*. An m-relationship connects a number of m-objects from different dimensions and allows to define *measures* at a specific level of granularity.

M-relationships reflect the dimensional structure of the m-cubes they are defined upon. A three-dimensional m-cube defines three-dimensional m-relationships to hold measures. Thus, the object types representing m-relationships in the Oracle object-relational database are dynamically created at run-time.

M-relationships always belong to a particular m-cube. M-cubes therefore provide functions to add and retrieve m-relationships. Each m-relationship in turn defines a number of functions and procedures to handle measures and metadata.

3.2.1 Adding m-relationships to m-cubes

M-relationships are strictly bound to a particular m-cube. The dimensions of an m-relationship correspond to the possessing m-cube's dimensions. Consequently, m-relationships of a particular m-cube necessarily connect m-objects that belong to the m-cube's constituting dimensions.

Each m-cube maintains its own table of m-relationships. All m-relationships of a particular m-cube are stored in the same table. These tables loosely correspond to *fact tables* in traditional (ROLAP) data warehouses, albeit with the actual facts stored in separate tables. However, they are not equal to *fact tables* and are thus referred to as m-relationship tables.

Since m-relationships are strictly dependent on a particular m-cube, their object type representation is created at the time of the m-cube's creation. Thus, for each concretized m-cube object type there exists exactly one m-relationship object type. The naming convention for m-relationship types follows a particular pattern: `mrel_`⟨*name of m-cube*⟩`_ty`, e.g., `mrel_sales_cube_ty`. Again, the naming scheme imposes a limitation on the length of the m-cube's name due to the Oracle database limit for entity names of 30 bytes.

M-relationships are identified within an m-cube by coordinates (attribute `coord`). Coordinates are represented as objects of type `coord_*_ty` which is dynamically created at run-time. The same naming scheme is employed for the coordinate object type as is for m-relationship object types: `coord_`⟨*name of m-cube*⟩`_ty`, e.g., `coord_sales_cube_ty`. A coordinate stores in its attributes the references to the m-objects linked together by the m-relationship it belongs to. The number of attributes corresponds to the number of the m-cube's – and thus the m-relationship's – dimensions. Each attribute contains a reference to a concretization of `mobject_ty`. For instance, the coordinate of a three-dimensional sales m-cube might hold references to m-objects of types `mobject_product_dim_ty`, `mobject_time_dim_ty`, and `mobject_location_dim_ty`. Each m-cube must contain an m-relationship that corresponds to its root-coordinate.

The concretized m-cube types define a function for adding m-relationships. Function `create_mrel` takes references to the m-objects that shall be linked by the m-relationship. The number of parameters varies between m-relationships of different m-cubes and cor-

responds to the number of the m-cube's dimensions. The parameters of the function are references to concretizations of `mobject_ty`. This ensures type safety and improves the performance over generically implemented object types.

Function `create_mrel` automatically inserts the newly added m-relationships into the m-cube's m-relationship table. No SQL `INSERT` statement needs to be issued by the user. The m-relationship table is an object table of the m-cube's m-relationship object type. Therefore, m-relationships cannot be inserted into an m-relationship table other than that of the m-cube they have been originally defined upon. Doing so will cause an error.

An example of how m-relationships are added to m-cubes is shown in listing 3.13. Consider the three-dimensional *sales* m-cube consisting of dimensions *product*, *time* and *location*. The m-cube object is retrieved by the SQL `SELECT` statement in line 13. Note that a type cast is necessary since the `create_mrel` function is only defined by the concretized subtypes of `mcube_ty`. Line 31 demonstrates how the m-relationship is actually added to the m-cube. The references to the m-objects connected by the m-relationship are passed as arguments to the function. Note that the references to the m-objects are references to the concretized m-object, one for each of the m-cube's dimensions. The order of the dimensions in the parameter list corresponds to the order of the dimensions in the list that has been passed to function `create_mcube` of package `mcube` when creating the m-cube. The same is true for all other functions and procedures whose parameter list is adapted to the number of dimensions: The order of the parameters depends on the order of the dimensions that were passed to the m-cube creation function. Function `create_mrel` alternatively accepts the names of the connected m-objects as shown in listing 3.13 line 37.

Coordinates and consequently m-relationships are partially ordered. The parent m-relationships of a particular m-relationship are not explicitly stated at the time of its creation. Rather, the ancestors are given implicitly through the notion of partial order of coordinates. Parent m-relationships are thus m-relationships with coordinates that are proper super-coordinates of another m-relationship's coordinate. A new m-relationship's coordinate must be a descendant of the m-cube's root-coordinate. The coordinate object type defines an order method `compare_to` that calculates the relative order of two coordinates. The notion of partial order of connection-levels is explained in chapter 2. The `compare_to` function of coordinates returns a value of -1 if the coordinate is a proper sub-coordinate of the argument coordinate. A value of 1 is returned in case that the argument coordinate is a proper sub-coordinate of the coordinate of whom the function is called. A value of 0 is returned in any other case. Only coordinates of the same object type, i.e., the same m-cube, can be compared. The function is thus defined as follows, with $(o_1, \ldots, o_n)$ and $(o'_1, \ldots, o'_n)$ denoting coordinates of the same n-dimensional m-cube:

$$(o_1, \ldots, o_n).compare\_to((o'_1, \ldots, o'_n)) := \begin{cases} -1 & \text{iff } (o_1, \ldots, o_n) \prec (o'_1, \ldots, o'_n) \\ 1 & \text{iff } (o_1, \ldots, o_n) \succ (o'_1, \ldots, o'_n) \\ 0 & \text{otherwise} \end{cases}$$

Listing 3.13: Adding m-relationships to the *sales* m-cube

```
1  DECLARE
2    sales_cube mcube_sales_cube_ty;
3
4    ref_product_dim REF mobject_product_dim_ty;
5    ref_time_dim REF mobject_time_dim_ty;
6    ref_location_dim REF mobject_location_dim_ty;
7
8    mrel mrel_sales_cube_ty;
9  BEGIN
10   /* select the sales cube from the mcubes table
11      NOTE: type cast is necessary to access the
12      create_mrel function */
13   SELECT TREAT(VALUE(mc) AS mcube_sales_cube_ty)
14   INTO   sales_cube
15   FROM   mcubes mc
16   WHERE  mc.cname = 'sales_cube';
17
18   /* retrieve the m-objects that shall be linked
19      together by the m-relationship */
20   SELECT REF(o) INTO ref_product_dim
21   FROM product_dim o WHERE o.oname = 'Product';
22
23   SELECT REF(o) INTO ref_time_dim
24   FROM time_dim o WHERE o.oname = 'Time';
25
26   SELECT REF(o) INTO ref_location_dim
27   FROM location_dim o WHERE o.oname = 'Location';
28
29   /* add an m-relationship at the sales cube's
30      root-coordinate */
31   mrel := sales_cube.create_mrel(ref_product_dim,
32                                  ref_time_dim,
33                                  ref_location_dim);
34
35   /* alternative signature taking m-object
36      names instead of references */
37   mrel := sales_cube.create_mrel('Car',
38                                  'Year2010',
39                                  'Switzerland');
40 END;
```

M-relationships are identified by their coordinate. The user has two possibilities for retrieving a particular m-relationship object from the m-cube's m-relationship table. First, the user can issue an SQL `SELECT` statement, specifying the m-relationships coordinate in the `WHERE` clause. Second, function `get_mrel` is provided by an m-cube to retrieve an m-relationship. The function takes the names of the m-objects connected by the m-relationship as arguments and returns the m-relationship.

3.2.2 Handling measures in m-relationships

M-relationships describe facts in hetero-homogeneous m-cubes. Facts represent real-world events of interest (e.g., sales) whereas *measures* describe these facts (e.g., the sold quantity or the revenues) [GMR98]. Unlike facts in the *Dimensional Fact Model*, facts in m-cubes can be heterogeneous in the sense that different sub-cubes may define the same measures at different levels of granularity. For example, revenues might be defined at a more specific granularity for sales in Switzerland compared to the sales in Austria. Furthermore, different measures may also have different levels of granularity. For instance, the quantity sold might be defined at a different level of granularity than measure revenue. The full details are explained in chapter 2.

In hetero-homogeneous m-cubes, the level of granularity of a particular measure is represented by its *connection-level*. A measure's connection-level is modeled using objects. The connection-level object type is dynamically created, alongside the m-relationship and the coordinate object types. Each attribute represents a level of one of the m-objects linked together by the m-relationship. Thus, the number of attributes of the connection-level type corresponds to the number of dimensions that constitute the m-cube. The naming scheme for the dynamically created object types follows the usual pattern: `conlevel_`⟨*name of m-cube*⟩`_ty`.

Similar to m-objects, each m-relationship has a *top-connection-level*. An m-relationship's top-connection-level is obtained by taking the top-level of each of the referenced m-objects [NST10]. In the prototypical implementation, the m-relationship object types provide a function `get_toplevel` which retrieves the top-connection-level of an m-relationship. M-relationships only provide values for measures – apart from default values and metadata – at a connection-level that is equal to the m-relationship's top-connection-level.

Procedure `add_measure` is provided by the m-relationship object types in order to define measures. The procedure takes as arguments the measure's unique name (`VARCHAR2`), its data type (`VARCHAR2`) and the value (`ANYDATA`). The procedure takes care of initializing the measure table, i.e., the table where the values for this measure are held, and stores this table's name within a nested table attribute of the m-relationship. The changes in the m-relationship's nested table are automatically persisted in the database, i.e., the entry in the m-cube's m-relationship table is updated.

M-relationships inherit measures from their ancestor m-relationships in the same manner as m-objects inherit attributes from their ancestor m-objects. A measure defined by an ancestor m-object must not be added to the descendant m-object again. The system implicitly handles inheritance. Adding an already inherited measure to the descendant

Method	Parameters	Return	Description
`add_measure`	name `VARCHAR2`, level `conlevel_*_ty`, data type `VARCHAR2`	–	Introduce a new measure at the specified level.
`set_measure`	name `VARCHAR2`, value `ANYDATA`	–	Set the asserted value for measures.
`set_measure`	name `VARCHAR2`, meta level `VARCHAR2`, default? `BOOLEAN` value `ANYDATA`	–	Set metadata or the value for an added measure.
`get_measure`	name `VARCHAR2`	`ANYDATA`	Get the value for the specified measure.
`delete_measure`	name `VARCHAR2`	–	Remove the specified measure from the m-relationship.
`has_measure`	name `VARCHAR2`, toplevel? `BOOLEAN`, introduced? `BOOLEAN` description `OUT measure_*_ty`	`BOOLEAN`	Look if the m-relationship defines a measure and output its description.
`has_measure`	name `VARCHAR2`, toplevel? `INTEGER`, introduced? `INTEGER`	`INTEGER`	SQL-compatible signature of `has_measure`.
`list_measures`	toplevel? `BOOLEAN`, introduced? `BOOLEAN`	`measure_*_tty`	List introduced or inherited measures.

Table 3.4: Measure handling member methods of m-relationships

m-relationship will result in an error. An m-relationship must provide values for measures that have been defined at the m-relationship's top connection-level. For instance, measure `revenue` might be defined at connection level $\langle$*model*, *month*, *city*$\rangle$ by the m-relationship connecting m-objects *Product*, *Time*, and *Location* in the sales m-cube. A descendant m-relationship between m-objects *FiatPunto55*, *Jan2010*, and *Vienna* connects m-objects of a more specific granularity with top-connection-level $\langle$*model*, *month*, *city*$\rangle$. It therefore provides a value for measure *revenue*. The measure must not be added again to the m-relationship between m-objects *FiatPunto55*, *Jan2010*, and *Vienna*.

Procedure `set_measure` is provided by the m-relationship object types in order to set measure values. The procedure takes the name of the measure to be set and its value (`ANYDATA`). Listing 3.14 demonstrates how measures are added to m-relationships and how an m-relationship at a more specific connection-level provides the measure's (asserted) value. Once set, a measure's value can be retrieved by invoking the m-relationship's `get_measure` function, which takes the measure's name as argument and

returns the asserted value as `ANYDATA`. This can only be done for measures at the m-object's top-level, as only for those measures an asserted value is given.

Listing 3.14: Defining and setting measures in m-relationships

```
1  DECLARE
2    sales_cube mcube_sales_cube_ty;
3
4    mrel mrel_sales_cube_ty;
5  BEGIN
6    (...)
7
8    /* get the m-relationship at highest connection-level */
9    mrel := sales_cube.get_mrel('Product', 'Time', 'Location');
10
11   /* define measure revenue at connection-level
12      model, month, city */
13   mrel.add_measure('revenue',
14                    conlevel_sales_cube_ty('model',
15                                           'month',
16                                           'city'),
17                    'NUMBER');
18
19   /* get a descendant m-relationship at top connection-
20      level model, month, city */
21   mrel := sales_cube.get_mrel('FiatPunto55',
22                               'Jan2010',
23                               'Vienna');
24
25   /* provide a value for the measure */
26   mrel.set_measure('revenue',
27                    ANYDATA.convertNumber(1500000));
28 END;
```

Whether an m-relationship defines or inherits a particular measure is checked by invoking the m-relationship object type member function `has_measure`. Measures introduced by ancestor m-relationships, i.e., m-relationships at a proper super-coordinate, are inherited. The query can be restricted such that only measures at the m-relationship's top-connection-level and/or only measures that have been introduced by the m-relationship itself are considered. The function returns `TRUE` if the measure is contained within the m-relationship, provided the restrictions are met; otherwise, `FALSE` is returned. A description of the measure is returned as output parameter (`measure_*_ty`).

A measure is further described by a dedicated object type. Each m-cube defines its own measure description type which is dynamically defined when the m-cube is cre-

ated: `measure_`⟨*name of m-cube*⟩`_ty`, e.g., `measure_sales_cube_ty`. The measure type contains the measure's name (attribute `measure_name`), the measure's connection level (attribute `measure_lvl` of type `conlevel_*_ty`) and attributes for describing the measure's data type, length and scale. A table type is defined for each measure description type (`measure_*_tty`).

A list of all measures of a particular m-object can be obtained by calling the `list_measures` function of the m-relationship. As with the `has_measure` function, the query can be restricted to measures at the m-relationship's top-connection-level and/or measures introduced by the m-relationship. The function returns a collection of measure descriptions (type `measure_*_tty`). Notice that if the `introduced_only` option is checked for a query, functions `has_measure` and `list_measures` consider only measures which have been originally introduced by the respective m-relationship – with one exception though. If a measure is moved to a more specific level of granularity it will be considered by these functions as `introduced_only` even though conceptually, this measure was originally introduced by an ancestor m-relationship.

Once a measure is introduced at an m-relationship by calling procedure `add_measure`, a subsequent introduction of a measure with the same name will cause an error (unique induction rule for measures). However, there is one exception to that rule: descendants of the m-relationship that introduced the measure can move the measure to a more specific level of granularity. For instance, measure *revenue* is originally introduced by the m-relationship at the cube's root-coordinate between m-objects *Product*, *Time*, and *Location* at connection-level $\langle model, month, city \rangle$. The revenue for car sales of the year 2010 in Switzerland in contrast might be measured at a more specific level of granularity. Consequently, the m-relationship connecting m-objects *Car*, *Switzerland*, and *Year2010* might move measure *revenue* to connection-level $\langle model, month, store \rangle$ (see listing 3.15 line 15).

Moving a measure to a more specific level of granularity is performed by executing the concretizing m-relationship's `add_measure` procedure as usual. The procedure will not cause an error if (i) the m-relationship is a concretization – i.e., at a sub-coordinate – of the m-relationship that originally introduced the measure, and if (ii) the new connection-level is a sub-connection-level of the previous connection-level. A connection-level $(l_1, \ldots, l_n)$ is the sub-connection-level of another connection-level $(l'_1, \ldots, l'_n)$ – written as $(l_1, \ldots, l_n) \prec (l'_1, \ldots, l'_n)$ – if, and only if, each level $l_i \in D_i$ is a sub-level of or equal to $l'_i \in D_i$ in the respective levels' dimension D_i and at least one level in l_i is a sub-level of l'_i in the levels' dimension D_i. A level is contained in a dimension ($l \in D$) if it is contained in the dimension's global level-hierarchy, i.e., at least one m-object of dimension D has the particular level l. For instance, the call to `add_measure` in listing 3.15 line 24 will cause an error since – even when the previous call to `add_measure` in the same listing is ignored – the user is trying to move measure *revenue* to a less specific level of granularity. For all descending m-relationships, the measure is defined at the new level of granularity. Functions `has_measure` and `list_measures` will only include the measure at the more specific level of granularity in their results. The partial order of connection levels can be checked by `ORDER` function `compare_to` of object type `conlevel_*_ty` which is defined by the following:

Listing 3.15: Moving a measure to a more specific connection-level

```
1  DECLARE
2    sales_cube mcube_sales_cube_ty;
3
4    mrel mrel_sales_cube_ty;
5  BEGIN
6    (...)
7
8    /* get the m-relationship representing the car sales
9       of the year 2010 in Switzerland */
10   mrel :=
11     sales_cube.get_mrel('Car', 'Year2010', 'Switzerland');
12
13   /* move measure revenue to a more specific level of
14      granularity: model, month, store */
15   mrel.add_measure('revenue',
16                    conlevel_sales_cube_ty('model',
17                                           'month',
18                                           'store'),
19                    'NUMBER');
20
21   /* the following call violates the unique induction
22      rule for measures and would cause an error if
23       called instead of the above statement */
24   mrel.add_measure('revenue',
25                    conlevel_sales_cube_ty('model',
26                                           'month',
27                                           'country'),
28                    'NUMBER');
29 END;
```

$$\langle l_1, \ldots, l_n \rangle .compare\_to(\langle l'_1, \ldots, l'_n \rangle) := \begin{cases} -1 & \text{iff } \langle l_1, \ldots, l_n \rangle \prec \langle l'_1, \ldots, l'_n \rangle \\ 1 & \text{iff } \langle l_1, \ldots, l_n \rangle \succ \langle l'_1, \ldots, l'_n \rangle \\ 0 & \text{iff } \langle l_1, \ldots, l_n \rangle = \langle l'_1, \ldots, l'_n \rangle \\ 9999 & \text{otherwise} \end{cases}$$

3.2.3 Handling metadata about measures

Measures can be further described by metadata. First, the aggregation function of the measure may be described using metadata. Second, measure units are specified by setting metadata, which is similar to setting metadata for m-object attributes. In general, the same rules are applied for handling metadata about measures as there are for handling metadata about an m-object's attributes.

Procedure `set_measure` defines an alternative signature which is used to set metadata. The alternative version of the `set_measure` procedure takes four arguments. First, the measure's name must be specified. Second, the meta level needs to be specified (`VARCHAR2`). Third, it has to be specified whether the value to be set is default or shared, i.e., whether the value can or cannot be altered by descendant m-relationships. Last, the value is passed as type `ANYDATA`.

As with m-object attribute metadata, the four-parameter `set_measure` procedure might also be used to set the measure's value at meta level `NULL`. In this case, a call to the four-parameter `set_measure` procedure corresponds to calling the two-parameter procedure.

The preferred way to represent metadata is the use of an m-object hierarchy. The use of plain information to represent metadata follows the same approach that is applied on m-object attribute metadata. It is straight-forward and will not be covered here. In the conceptual model, measures are provided with an aggregation function and a unit of measurement. In the logical implementation, this information is represented as metadata. The final semantics is to be implemented yet. It is very likely that the use of m-object hierarchies for representing measure metadata will be mandatory in future versions. However, in the current version, the prototypical implementation relies on the use of plain data to define a measure's aggregation function.

The proposed approach to represent measure metadata through m-object hierarchies generally follows the approach for representing m-object attribute metadata. Furthermore, measures demand an aggregation function to be specified in addition to a measure unit. Object type `mrel_*_ty` provides the procedure `set_aggregation_function` for the purpose of defining a measure's aggregation function. The aggregation function is internally stored as measure metadata. It may thus also be set using procedure `set_attribute`. The internal representation of measure metadata has not yet been ultimately fixed. It is thus safer for the user to rely on the `set_aggregation_function` procedure which provides an interface for setting the aggregation function that is (more) independent of the internal representation. The definition of a measure's aggregation

function must follow specific semantics so that the system can find this information. Procedure `set_aggregation_function` demands the user to pass the string representation of the desired aggregation function, i.e., SUM, MIN, MAX. This string representation has to be converted to `ANYDATA` since future versions may rely on the use of m-object hierarchies to represent aggregation functions. Function `get_aggregation_function` returns the string representation of a measure's aggregation function – converted to `ANYDATA`. If no aggregation function is explicitly specified – i.e., function `get_aggregation_function` returns `NULL` – all operations assume a measure's aggregation function to be SUM.

Listing 3.16 illustrates how the aggregation function for measure *revenue* can be specified in the current prototype. The aggregation function of a measure cannot be changed by descendant m-relationships, i.e., it is a shared value. The `set_aggregation_function` procedure handles the task of inserting the information into the `measure_metadata` nested table transparently.

Listing 3.16: Defining a measure's aggregation function

```
1  DECLARE
2    sales_cube mcube_sales_cube_ty;
3
4    mrel mrel_sales_cube_ty;
5  BEGIN
6    (...)
7
8    /* get the m-relationship at highest connection-level */
9    mrel := sales_cube.get_mrel('Product', 'Time', 'Location');
10
11   /* define measure revenue at connection-level
12      model, month, city */
13   mrel.add_measure('revenue',
14                    conlevel_sales_cube_ty('model',
15                                           'month',
16                                           'city'),
17                    'NUMBER');
18
19   /* the aggregation function of measure 'revenue' is SUM */
20   mrel.set_aggregation_function
21     ('revenue', ANYDATA.convertVarchar2('SUM'));
22 END;
```

Future implementations might rely on an m-object hierarchy to represent the aggregation functions for measures. This is done by extending the already introduced metadata m-object hierarchy (figure 3.2). Under m-object *Metadata*, m-object *Aggregation* with top-level *metadata* and level *function*. Each descendant of m-object *Aggregation* at top-level *function* represents an aggregation function. The advantage of this approach is that

it would allow for the specification of metadata for aggregation functions. This approach might be employed in future versions.

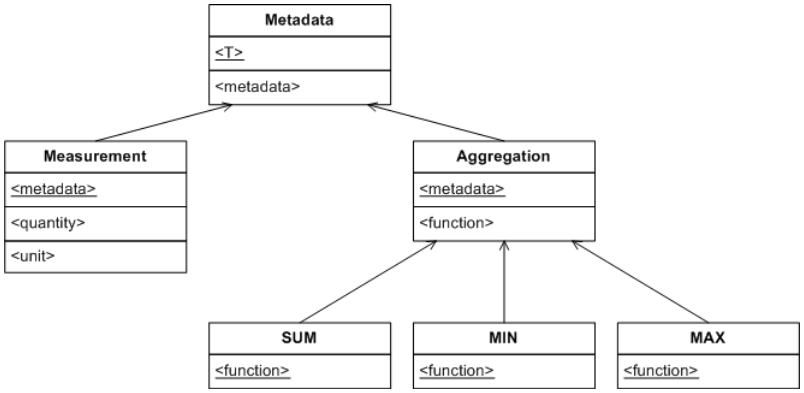


Figure 3.2: Proposed structure of the *metadata hierarchy* extended for the representation of aggregation functions

3.3 OLAP with hetero-homogeneous hierarchies and cubes

Without a suitable query algebra, the information that is contained within a data warehouse remains unexploited. In this case, the whole exercise of building and keeping a data warehouse becomes useless. After all, query operations represent the core feature of a data warehouse. The extension package consequently provides a prototypical implementation of the m-cube query algebra.

The operations of the query algebra are grouped in three main categories. First, operations are provided to convert a hetero-homogeneous data warehouse based on m-cubes into a flat ROLAP data warehouse. On the traditional ROLAP data structures, SQL queries can be performed to extract and analyze data. Second, query operations are provided that take an m-cube as input and produce an m-cube as output in order to reduce the cube size and narrow the amount of data. Third, operations are given for extracting facts and measures from a cube. The section concludes with an explanation of query views that can be used to reduce time and space complexity of the query operations.

The prototypical m-cube data warehouse system is constructed atop an object-relational database. In many ways, the implementation makes use of object-oriented features. In object-oriented databases *object-preserving* and *object-generating query semantics* – as opposed to the *relational semantics* in relational databases – have to be distinguished [SS91]. It is thus necessary to define whether an m-cube query operation is object-preserving, i.e., parts of the input objects are included in the result, or object-generating, i.e., new objects are created. On the other hand, some of the m-cube query operations produce relations and not objects.

3.3.1 Creating flat data warehouse schemata

The integration of hetero-homogeneous data warehouses into an existing data warehouse infrastructure is facilitated through operations that allow the export as ROLAP data warehouse tables. Data warehouses are often logically represented as relational database tables. The migration from traditional implementations to a hetero-homogeneous form of data organization is simplified by suitable export operations. Existing algorithms and query expressions tailored to the ROLAP data organization can thus be re-used for hetero-homogeneous data warehouses if export functionality is provided by the data warehouse software.

The most common ROLAP structures are the star and the snowflake schema. Both are widely used logical data warehouse schemata. Both schemata differentiate between fact and dimension tables. Fact tables store the measures whereas the dimension tables hold non-dimension attributes. Hetero-homogeneous data warehouses can be transformed into star or snowflake schemata. However, those data structures were not originally designed for containing heterogeneities and are in many ways ill-suited for this breed of data warehouses. It is therefore necessary to adapt the schemata in order to represent hetero-homogeneous hierarchies and cubes. A detailed description of the produced star and snowflake schemata is included in chapter 4.

Star schema

The hetero-homogeneous data warehouse can be transformed into a ROLAP data warehouse based on the star schema. Even though not originally conceived for heterogeneous data warehouses, the star schema can nevertheless be used to represent hetero-homogeneous hierarchies and facts.

The star consists of one dimension table for each of the m-cube's dimensions and a fact table to represent the measures. The m-cube object type provides the function `create_star` to export its data in the form of a ROLAP star schema. Likewise, each dimension provides a function `create_star` that creates a dimension table containing non-dimensional information contained in its m-objects.

The `create_star` procedure of dimension objects allows the user to indirectly specify the dimension table's name through passing a suffix string that is appended to the name of the dimension. If the user specifies *$star* as suffix, the star representation of dimension *product_ dim* is stored in table `product_dim$star`. Furthermore, the user may specify whether the dimension table should be homogeneous – i.e., only attributes shared by all m-objects with the same top-level are contained – or heterogeneous. For example, consider a heterogeneous *product* dimension with root m-object *Product*. The root m-object defines attribute *costs* at level *model*. Under *Product*, m-objects *Car* and *Book* are defined at the *category* level. M-object *Car* introduces an additional attribute *maxSpeed* at the *model* level. Only cars assert a value for this attribute. If this hierarchy is to be transformed into a heterogeneous dimension table following the star schema, tuples representing m-objects under *Book* will contain `NULL` values in the column containing the values for attribute *maxSpeed*.

Procedure `create_star` of m-cube objects assumes the task of creating a dimension table according to the star schema for each of the m-cube's dimensions. The procedure furthermore creates a fact table that holds the measure values. The user can specify a suffix string that is appended to the m-cube's name and is used as the suffix for the m-cube's dimension tables. The user has to ensure that no duplicate table names occur with the specified suffix string.

An m-cube's fact table according to the star schema can hold heterogeneities. In order to achieve this, a common approach for storing aggregates within the star schema is adapted. A level attribute is introduced in the dimension tables: this attribute specifies the aggregation level of the tuple. A detailed description of this approach can be found in chapter 4.

Snowflake schema

The snowflake schema arranges the non-dimensional data within a normalized table hierarchy consisting of multiple dimension tables for each of the m-cube's dimensions. The snowflake schema is ill-suited for representing heterogeneities. The data organization thus has to be adapted for the representation of hetero-homogeneous dimension hierarchies. A detailed description of the logical structure of the employed snowflake schema can be found in chapter 4.

While the star schema defines only one dimension table for each of the m-cube's dimensions, the snowflake schema consists of a table hierarchy with one table for each m-object level within the dimension. Heterogeneities in the dimension's level-hierarchy need to be overcome by introducing *dummy* tuples. For example, the *brand* level is introduced under level *category* by m-object *Car*. Consequently, m-objects under m-object *Book* do not have level *brand*. The snowflake schema defines one table for each level. Thus, the *product* dimension – with levels *top*, *category*, *brand*, and *model* – defines four dimension tables. Each table corresponds to a particular level and references the tables of its parent levels. Dummy tuples are inserted for heterogeneous aggregation paths. For instance, the dummy brand *Book* is inserted into the dimension table for the *brand* level. The entries in the *model* level dimension table representing m-objects with top-level *model* that are descendants of *Book* will reference the dummy entry in the *brand* dimension table.

In contrast to the star schema, the snowflake schema organizes the measures of the m-cube in multiple fact tables. Heterogeneities in the level of granularity of measures cannot be represented in a single fact table. The fact tables are organized according to the *fact constellation* schema (cf. [EN07], p. 984). Each of the generated fact tables contains measures at a specific level of granularity.

3.3.2 Branching dimensions

Object type `dimension_ty` provides the `branch` function that allows the user to extract all descendant m-objects of a specified root m-object and thus reduce the size of a dimension; the selected set of m-objects is the source for building a new dimension. The function builds a new dimension with a different root m-object that is completely independent of

the source dimension. Independent copies of the source dimension's m-objects are added to the new dimension. Note that no attributes introduced by ancestor m-objects of the new dimension's root are lost. Attributes that have been introduced by an ancestor of the new root m-object are introduced by the branch dimension's root m-object. Of course, the new root m-object must define the level for which the attribute has been added – which might not be the case when an m-object has multiple parent m-objects. The `branch` function takes as arguments the name of the m-object that will be the new dimension's root and the dimension name of the new branch. The thus obtained dimension is automatically inserted into the `dimensions` table and can be used to construct independent data marts. The `branch` function of dimensions is object-generating in the sense that new dimension objects independent of the source are created. The result is a consistent, stand-alone dimension holding independent copies of the source dimension's m-objects. The `branch` function is a prerequisite for performing closed object-generating queries on m-cubes.

The usage of the *dimension branch* in the extension package is illustrated in listing 3.17. The `branch` function of dimensions takes the name of the new dimension's root m-object, in this case m-object *Alps*. A new dimension object is created and transparently added to the `dimensions` table. The name of the result dimension is specified by the user as the second argument to the `branch` function. The user has to be aware of the fact that the name of the branch must not be used by another dimension, otherwise an error will be thrown. The dimension branch is a full dimension object that can be referenced by m-cubes. Changes to the new dimension will consequently have no effect on the source dimension.

Not all m-objects that are descendants of the root m-object are included in the result dimension. During the execution process, each of the selected m-objects from the original dimension is transformed in order to create an independent copy of the original m-object. Some levels are disregarded in the copies of the original m-objects, e.g., level *store* of the source dimension *location* is not included for it has been originally introduced by an m-object not included within the result dimension. M-objects whose top-level was originally introduced by an m-object not included in the result dimension are furthermore dropped, e.g., m-objects with top-level *store* in the *location* dimension.

3.3.3 Object-generating closed m-cube query operations

Closed m-cube query operations are performed on m-cubes and return a new m-cube as result. The *dice* operation extracts a sub m-cube from a given m-cube, comprising only coordinates that are descendants of a specified coordinate. *Projection* refers to the reduction of measures and m-relationships such that only a specified set of measures is contained in the new m-cube. Finally, the *slice* operation allows to select only coordinates of an m-cube that satisfy certain conditions. The closed query operations serve the purpose of reducing the set of measures and coordinates. With regard to m-cubes and m-relationships held by the m-cube, they are generally object-generating in the sense that they create a new m-cube object with independent m-relationship copies. With

Listing 3.17: Performing a branch operation on a dimension

```
1  DECLARE
2    location_dim dimension_ty;
3    alps_dim dimension_ty
4  BEGIN
5    /* select the source dimension */
6    SELECT VALUE(dim) INTO location_dim
7    FROM   dimension dim
8    WHERE  dim.dname = 'location_dim';
9
10   /* perform the branch operation */
11   alps_dim :=
12     location_dim.branch('Alps',
13                         'alps_dim');
14 END;
```

regard to dimensions they are either object-preserving or object-generating in the sense that new dimension objects with independent m-object copies are optionally created.

The object-generating nature of the closed m-cube query operations allows for the creation of data marts with a reduced set of m-relationships and measures. This approach is time- and space-consuming and the query operations provided by the m-cube should thus only be employed if the creation of an independent data mart is intended. For object-preserving query operations the user is referred to *query views*.

Projection

A projection is applied on an m-cube to reduce its set of measures. The user specifies a set of measures from the source m-cube which is to be included in the result m-cube. Function `project` in `mcube_*_ty` is used to perform a projection on an m-cube. The function returns a new m-cube that references the same dimensions as the source m-cube. The result m-cube furthermore has the same root-coordinate as the source m-cube. The m-relationships of the new m-cube, however, are independent copies of the original, including independent copies of the measure tables. The projection operation therefore preserves the dimension objects but is object-generating concerning the new cube's m-relationships and the m-cube object itself.

Function `project` takes the name of the result m-cube that is to be generated and a list of names of those measures of the source m-cube that are to be included in the result. The names are passed as `names_tty` which is a table type of `VARCHAR2`. The function returns the newly created m-cube object as result. Listing 3.18 illustrates how a projection can be performed on an existing sales cube. Notice that the variable containing the source m-cube object is of concretized type `mcube_sales_cube_ty` whereas the variable that is going to receive the projected m-cube is of the abstract super type `mcube_ty`. At the

time the projection is performed, type `mcube_project_cube_ty` does not yet exist. The type of the result m-cube is dynamically created and consequently the function returns an object of the abstract base type.

Listing 3.18: Performing a projection on an m-cube

```
1  DECLARE
2    included_measures names_tty;
3    sales_cube        mcube_sales_cube_ty;
4    projected_cube    mcube_ty;
5  BEGIN
6    (...)
7
8    /* include only measures 'revenue' and 'qtySold' */
9    included_measures :=  names_tty('revenue', 'qtySold');
10
11   /* perform the projection */
12   projected_cube :=
13     sales_cube.project('projected_cube', included_measures);
14 END;
```

Dice

The *dice* operation extracts a sub m-cube with a specified root-coordinate from a given m-cube. The resulting m-cube contains all m-relationships at sub-coordinates of the new m-cube's root-coordinate. The dimensions of the new m-cube are generally the same as the source m-cube's dimensions. The `dice` function provides the optional feature to branch the source m-cube's dimension and thus create a result m-cube with different dimensions. All descendant m-relationships of the specified root-coordinate are included in the result m-cube.

Since the *dice* operation produces an m-cube as output, m-relationships between the new root-coordinate's ancestor m-objects need to be transferred to another top-connection-level. For all m-relationships in the input m-cube, each m-object that is an ancestor of the respective dimension's new root m-object has to be replaced with the new root m-object. M-relationships that involve at least one m-object that is not in a concretization relationship with the new root m-object are not considered in the result m-cube. Furthermore, measures are only considered if their connection-level is contained in the new dimension, i.e., each level is contained in the respective m-object's level-hierarchy. The same principle basically applies for transferring m-relationships as for transferring attributes when branching dimensions.

Each concretization of `mcube_ty` defines a `dice` function that creates a new sub m-cube. The function takes as arguments the name of the resulting sub m-cube and the unique names of the m-objects that represent the new m-cube's root-coordinate. A boolean ar-

gument states whether the dimensions should remain the same – i.e., an object-preserving query (`dim_generating = FALSE`) with respect to dimension objects – or copied using the dimensions' branch function – i.e., an object-generating query (`dim_generating = TRUE`) with respect to dimension objects. The parameter list is not generic but dynamically adapted to the number of the source m-cube's dimensions. Listing 3.19 illustrates the function call for dicing an m-cube. The `dice` function for the three-dimensional *sales* m-cube takes the names of three m-objects that constitute the result m-cube's root-coordinate.

Listing 3.19: Applying a dice operation on an m-cube

```
1  DECLARE
2    source_mcube mcube_sales_cube_ty;
3    result_mcube mcube_ty;
4  BEGIN
5    (...)
6
7    /* retrieving a cube containing car sales in 2010 in
8       the Alps region in dimension object-preserving mode */
9    result_mcube :=
10     source_mcube.dice('car_sales', FALSE,
11                       'Car', 'Year2010', 'Alps');
12 END;
```

The `dice` function provided by `mcube_*_ty` is an object-generating query with respect to the m-cube and m-relationship objects. Independent copies of the m-cube object and all of its m-relationship objects are created. The `dice` function is either object-preserving or object-generating with respect to the dimension objects that are referenced by the source m-cube. The user may specify whether the dice operation should be *dimension object-preserving* or *dimension object-generating*. Dimension object-generating query semantics are not properly supported in the current version.

When calling function `dice` of `mcube_*_ty` some measures are not included in the result m-cube even though they are defined by an m-relationship at a descendant coordinate of the new root-coordinate. An example will clarify this issue which occurs when the *dice* operation is applied with object-generating query semantics. The problem is encountered both with dimension object-generating and dimension object-preserving queries. Consider the *sales* m-cube that is depicted in figure 3.3. Measure *revenue* is introduced at the root-coordinate at connection-level $\langle model, month, city \rangle$. Car sales in Switzerland in the year 2010, however, are measured at a more detailed level of granularity, namely $\langle model, month, store \rangle$. Furthermore, for car sales in Switzerland in 2010, an additional measure *qtySold* – the sold quantity – is available at connection-level $\langle model, month, store \rangle$. In turn, measure *cheapestOffer* is available for sales of all products in the Alps region in the year 2010. The m-relationships involving cities *Lausanne*

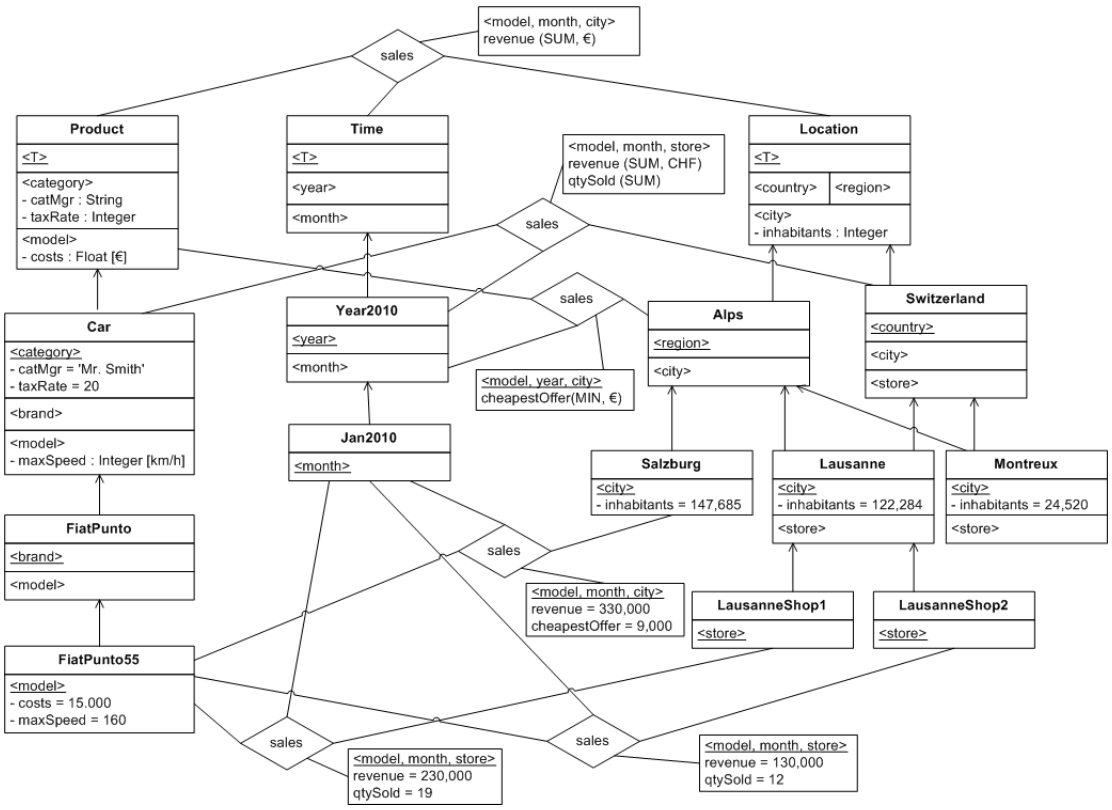


Figure 3.3: *Sales* m-cube used as source for applying a *dice*

and *Montreux* which assert a value for measure *cheapestOffer* are omitted due to space considerations.

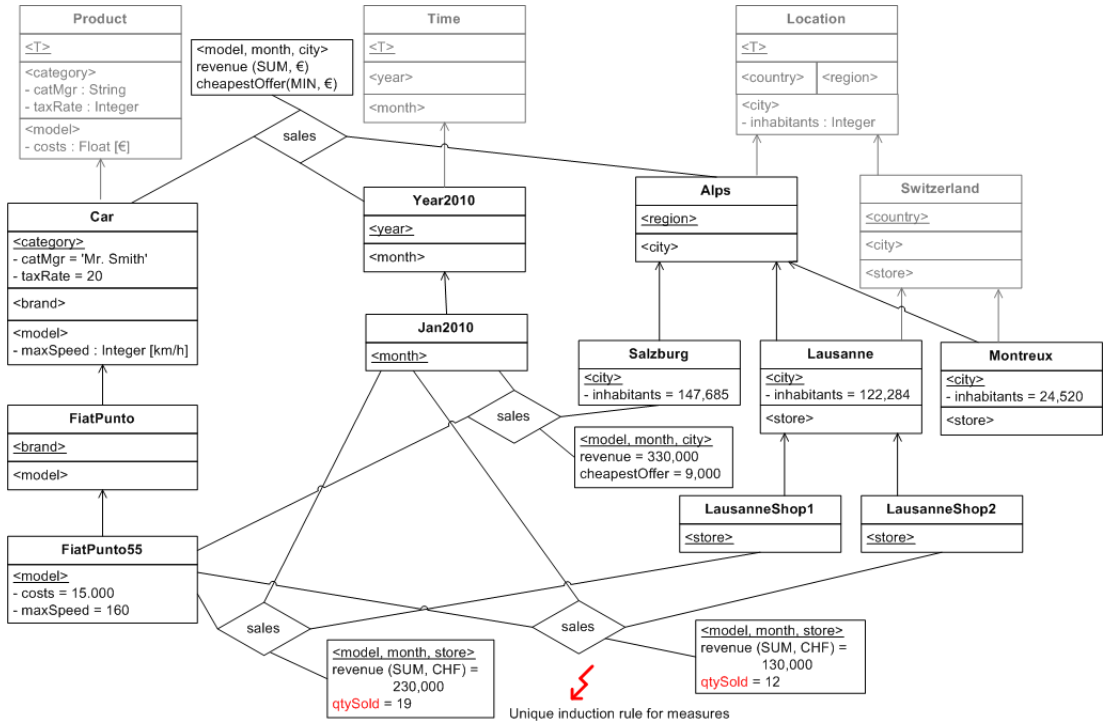


Figure 3.4: A sub-cube of the *sales* m-cube after applying *dice*

Some issues have to be considered when the *dice* operation is applied on the m-cube depicted in figure 3.3. Assume that only car sales in the *Alps* region in the year 2010 are of interest. The user consequently calls the `dice` function of the m-cube object. If the query is dimension object-preserving, the same dimension objects are used. The root-coordinate of the result m-cube, however, changes to (*Car*, *Year2010*, *Alps*). The result m-cube is depicted in figure 3.4. Notice that only m-relationships at sub-coordinates of the result m-cube's root-coordinate – (*Car*, *Year2010*, *Alps*) – are included within the result m-cube. Further notice that for each m-relationship that is an ancestor of the m-relationship at the new m-cube's root-coordinate, the coordinate is altered in order for the m-relationship to be included in the result m-cube. This is necessary in order to obtain a consistent m-cube; otherwise the *unique induction rule for measures* might be violated. Consider measure *revenue* defined by the m-relationship at the source m-cube's root-coordinate. This m-relationship is not included in the result m-cube. The m-relationships in the result m-cube that assign a value for this measure consequently violate the *unique induction rule*.

The m-relationship at the root-coordinate of the source m-cube cannot be included without being modified since its coordinate is an ancestor of the result m-cube's root-coordinate. Therefore, the m-relationship between m-objects *Product*, *Time*, and *Loca-*

tion is moved to coordinate (*Car*, *Year2010*, *Alps*). This solves the unique induction issue for measure *revenue*. The m-relationship between m-objects *Product*, *Year2010*, and *Alps* cannot be included in the result m-cube without being modified either; m-object *Product* is an ancestor of m-object *Car*. The result m-cube's equivalent to this m-relationship thus has m-object *Product* in its coordinate replaced with m-object *Car*. An m-relationship at coordinate (*Car*, *Year2010*, *Alps*) already exists; measure *cheapestOffer* is thus added to the existing m-relationship. Notice that the m-relationships involving m-objects *Lausanne* and *Montreux* between m-objects at top-level *month* of the *time* dimension and m-objects at top-level *model* of the *product* dimension are once again omitted in figure 3.4 due to space considerations.

The condition that an m-relationship at the result m-cube's root-coordinate must already exist is relaxed by the implementation (see chapter 2). The above procedure of altering the existing m-relationships in order to obtain a consistent m-cube necessarily creates an m-relationship at the new m-cube's root-coordinate. Since the consistent source m-cube must have an m-relationship defined at its root-coordinate, this m-relationship is moved to the result m-cube's root-coordinate in order to be able to consistently include the measures originally introduced by this m-relationship.

The m-relationship between m-objects *Car*, *Year2010*, and *Switzerland* of the source m-cube (figure 3.3) that introduces measure *qtySold* cannot be included in the result m-cube, neither without nor with modification. Since *Switzerland* is not an ancestor of m-object *Alps*, the m-relationship is disregarded completely. Consequently, measure *qtySold* cannot be introduced by this m-relationship. On the other hand, the m-relationships at top connection-level $\langle model, month, store \rangle$ which assert values for measure *qtySold* are descendants of the result m-cube's root-coordinate and can thus be included in the result m-cube. However, including measure *qtySold* violates the unique induction rule for measures. Consequently, the measure is not included in the result m-cube.

Another problem occurs when performing the *dice* operation in dimension object-generating mode which basically follows from the issue encountered when branching dimensions. Consider the example in figure 3.3 where m-object *Switzerland* defines an additional level *store* under *city* that is not defined by m-object *Alps*. The three-dimensional m-cube involving this dimension might define measure *revenue* at its root-coordinate at connection-level $\langle model, month, city \rangle$, whereas car sales in Switzerland would measure the revenue at connection-level $\langle model, month, store \rangle$. Performing a dimension object-generating *dice* on this m-cube to get a sub-cube containing only sales of the Alps region would result in level *store* being disregarded, since a new consistent dimension would be created using function `branch` (figure 3.5). In this case, measure *revenue* would get lost for car sales in Switzerland. The solution was to aggregate the revenues of the stores in Switzerland and thus move the measure's connection-level to $\langle model, month, city \rangle$. This solution is not supported by the current prototype.

Slice

The *slice* operation reduces the set of m-relationships by specifying selection criteria on the m-objects that constitute an m-relationship's coordinate. The selection of the

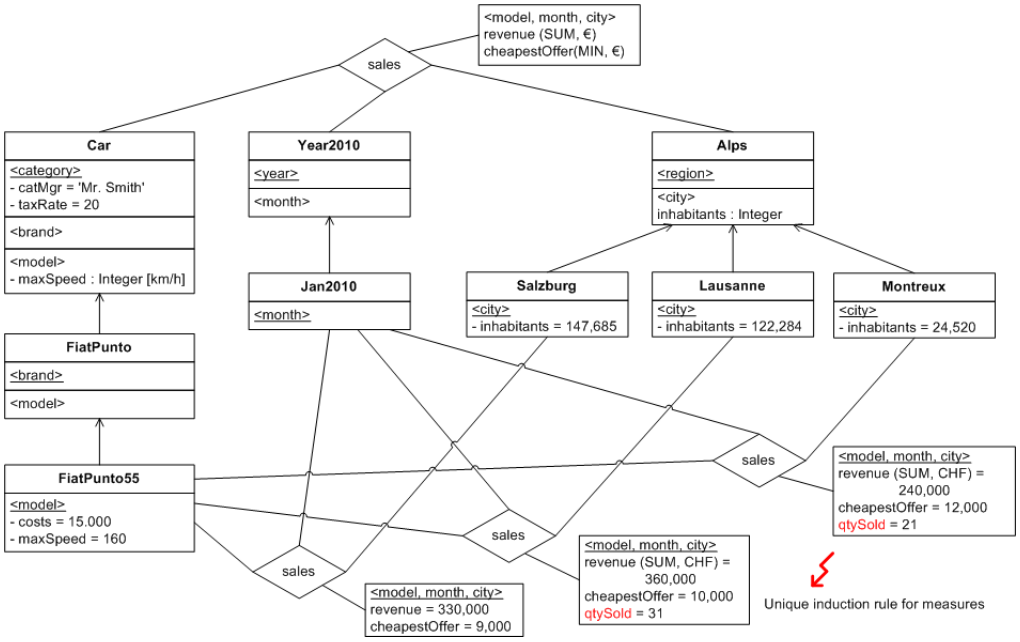


Figure 3.5: A sub-cube of the *sales* m-cube after applying *dice* in dimension object-generating mode

m-relationships is based on the attribute values of the m-objects that are connected by them. Only m-relationships whose connected m-objects satisfy certain criteria are added to the resulting m-cube. The resulting m-cube has the same dimensions and the same root-coordinate as the source m-cube (argument `dim_generating = FALSE`), or alternatively creates an m-cube with independent dimension objects using the `branch` function (argument `dim_generating = TRUE`). M-relationship objects of the result m-cube, however, are independent copies of the originals.

M-cubes implement the `slice` function which is dynamically concretized at the time of the m-cube's creation. For each dimension, the function takes a selection predicate as argument. A selection predicate is represented as an object of type `slice_predicate_ty`. A slice predicate is constructed for a particular m-object level, stored in attribute `lvl`. The constructor of `slice_predicate_ty` takes the predicate's level as argument. The user can add boolean expressions to the predicate by invoking procedure `add_expression`. This procedure takes as arguments the name of the attribute that should satisfy the criterion, a comparison operator (e.g., $>$, $<$, $=$, $<=$, $>=$) in SQL notation, and the value that is compared with the attribute's actual asserted value using the specified comparison operator. The added expressions are seen as a conjunction, i.e., an m-object has to satisfy all expressions in order to satisfy the predicate. The set of selected m-objects consists of the m-objects that satisfy the slice predicate as well as their ancestors and descendants. Only m-relationships between the selected m-objects are included in the result m-cube. The user is not obliged to give a slice predicate for

each of the m-cube's dimensions. In this case, all m-objects of the respective dimension are regarded.

Listing 3.20 illustrates how a *slice* operation might be performed on an existing *sales* cube to obtain only sales in big cities with a population that exceeds 100,000 inhabitants. First, a predicate is created containing the expressions that have to be satisfied by the m-objects of the *location* dimension. Function `slice` is then called with only a predicate for the *location* dimension while the predicates for the other dimensions are `NULL`. The second argument states that the original dimension objects should be referenced (*object-preserving*).

Listing 3.20: Applying a slice operation on an m-cube

```
1  DECLARE
2    sales_cube mcube_sales_cube_ty;
3    sliced_cube mcube_ty;
4
5    location_predicate slice_predicate_ty;
6  BEGIN
7    (...)
8
9    /* create a predicate for level 'city' */
10   location_predicate := slice_predicate_ty('city');
11
12   /* construct a predicate that selects only cities
13      where the asserted value of 'inhabitants'
14      is >= 100,000 */
15   location_predicate.
16     add_expression('inhabitants',
17                    '>=',
18                    ANYDATA.convertNumber(100000));
19
20   /* perform the slice in object preserving mode */
21   sliced_cube :=
22     sales_cube.slice('sliced_cube', FALSE,
23                      NULL, NULL, location_predicate);
24 END;
```

3.3.4 Fact extraction and aggregation of measures

Data warehouses allow for the retrieval of aggregated business data. Depending on the situation, the user might be interested in business data at a more or less general level of granularity. While measure *revenue* is stored at the level of granularity $\langle model, month, store \rangle$, e.g., the revenues of car model *FiatPunto55* in January 2010 in the city of *Salzburg*, the

user might be interested in aggregated sales figures, e.g., the revenues of all car models in the year 2010 in *Austria*. The prototype provides function `value_aggregation` for the purpose of retrieving aggregated measure values. Note that no string measures are supported by the `value_aggregation` function. Furthermore, conversion between measure units is not supported in the prototypical implementation but will be included in future releases. With an m-object hierarchy for representing measure units, the prototype could be extended with m-object methods in order to provide conversion functions.

Function `value_aggregation` returns an aggregated value of a specified measure. The function takes as arguments the name of the measure (`measure_name`) and a roll-up coordinate – either as an object of type `coord_*_ty` or by specifying the names of the coordinate's m-objects. The function considers all measure values given by the m-relationships defined at the roll-up coordinate or one of its sub-coordinates. For example, with roll-up coordinate (*Car*, *Year2010*, *Austria*) and argument `measure_name` being *revenue*, function `value_aggregation` returns the revenues of car sales in *Austria* in the year 2010; the example is illustrated in listing 3.21 line 13. The aggregation function defined for the respective measure is used to calculate the aggregate value. The aggregation of values function currently only works for measures defined for data type `NUMBER`.

Function `fact_extraction` builds on the aggregation of values. This function takes a roll-up coordinate and a set of measures and retrieves the aggregated values for the given measures at the given roll-up coordinate. The result is written into a newly created table; the name of table is specified by the user. Listing 3.21 line 21 illustrates the usage of the `fact_extraction` function. The function retrieves the sold quantity and the revenue of car sales in the year 2010 in Switzerland and stores the values into the newly created table `sales_fact` (table 3.5). The `sales_fact` table is a relation with only one tuple. For each measure, there is a column containing the aggregated value; `NULL` is contained if there was no asserted value at the given roll-up coordinate or below. The roll-up coordinate is stored within an object column of type `coord_*_ty`.

COORD			REVENUE	QTYSOLD
PRODUCT_DIM	TIME_DIM	LOCATION_DIM		
REF Car	REF Year2010	REF Switzerland	4,200,000	1,100

Table 3.5: Result relation of applying *fact extraction* on the *sales* m-cube

3.3.5 Object-preserving query views

The closed query operations that are applied on m-cubes create a new m-cube as result each time the operation is performed. This object-generating approach is suitable for creating independent data marts; it is rather inefficient for information extraction. Query views solve this problem by providing a fully object-preserving query mode. A query view is an object of type `queryview_*_ty` that manages a list of query expressions. This makes querying m-cubes using query views a two-step process. The query is first defined and afterwards evaluated.

Listing 3.21: Applying *fact extraction* on the *sales* cube

```
1  DECLARE
2    aggregated_value ANYDATA;
3
4    sales_cube mcube_sales_cube_ty;
5  BEGIN
6    /* retrieve the sales cube object */
7    SELECT VALUE(mc)
8    FROM   mcubes mc
9    WHERE  mc.cname = 'sales_cube';
10
11   /* get the value for measure 'revenue' rolled-up
12      to coordinate (Car, Year2010, Austria) */
13   aggregated_value :=
14       sales_cube.value_aggregation('revenue',
15                                    'Car',
16                                    'Year2010',
17                                    'Austria');
18
19   /* retrieve the values of measures 'revenue' and 'qtySold'
20      rolled-up to coordinate (Car, Year2010, Switzerland) */
21   sales_cube.fact_extraction('sales_fact',
22                              names_tty('revenue', 'qtySold'),
23                              'Car',
24                              'Year2010',
25                              'Switzerland');
26 END;
```

A query view is always m-cube specific. The object type `queryview_*_ty` is created dynamically together with the m-cube. An object of type `queryview_*_ty` is obtained by calling function `new_queryview` of the m-cube that is to be analyzed. A query view holds a reference to its base m-cube and maintains a list of query expressions that is to be evaluated sequentially.

The query view object type provides a number of functions and procedures to perform query operations on the referenced base m-cube. Unlike the query operations provided by `mcube_*_ty`, query definition and execution are two completely independent steps. First, the user adds a number of closed query expressions to the query view's expression list. For this end, `queryview_*_ty` basically provides the same interface for closed query operations as `mcube_*_ty`. For instance, function `dice` of a three-dimensional m-cube takes three m-object names, one for each of the base m-cube's dimensions. Instead of returning a new m-cube with a different set of m-relationships, function `dice` of `queryview_*_ty` adds a corresponding dice expression to its expression list.

Query expressions are represented as objects of type `expr_*_ty`. For each query operation – i.e., dice, slice, project – exists one concretization of `expr_*_ty`, namely `dice_expr_*_ty`, `slice_expr_*_ty`, and `project_expr_*_ty`. The expression objects are created by the query view's respective functions and are transparently added to the query view's expression list. Function `add_expression` can be used to add a previously constructed expression object rather than passing the query parameters to functions `slice`, `dice`, and `project`.

A query view object further maintains a set of references to the selected m-relationships of the base m-cube and a set of included measure names. The set of m-relationship references initially contains references to all of the base m-cube's m-relationships and measures. After the user has specified and added all the query expressions, the query can be evaluated by calling function `evaluate`. The function manipulates the query view's set of m-relationships and measures. After its evaluation, the query view's expression list is re-set and the query view in its current state can be used as the basis for further query operations. All of the query functions return the query view's `SELF` reference as result.

The query view provides a procedure for materializing its root-coordinate, the set of m-relationship references, and the set of measures within a table (procedure `materialize`). All the m-relationship references as well as the names of the selected measures contained within the query view are consequently dumped into a table with a user-defined name. Furthermore, the query view provides export functionality for flat data warehouse schemata and fact extraction. These procedures are similar to the corresponding procedures of the m-cubes, with the exception that only m-relationships and measures contained in the query view's respective sets are considered. Listing 3.22 illustrates the use of query views.

Method	Parameters	Return	Description
`dice`	D_1 m-object name `VARCHAR2` … D_n m-object name `VARCHAR2`	`SELF`	Append a dice expression to the expression list
`slice`	D_1 predicate `slice_predicate_ty` … D_n predicate `slice_predicate_ty`	`SELF`	Append a slice expression to the expression list
`project`	measure list `names_tty`	`SELF`	Append a projection expression to the expression list
`evaluate`	–	`SELF`	Evaluate expression list
`evaluate`	dice expression `dice_expr_*_ty`	`SELF`	Immediately evaluate a dice expression.
`evaluate`	slice expression `slice_expr_*_ty`	`SELF`	Immediately evaluate a slice expression.
`evaluate`	project expression `project_expr_*_ty`	`SELF`	Immediately evaluate a project expression.
`materialize`	table name `VARCHAR2`	–	Dump the view's set of m-relationships into a table.
`create_star`	suffix `VARCHAR2` homogeneous? `BOOLEAN`	–	Export into a ROLAP star.
`create_snowflake`	suffix `VARCHAR2` homogeneous? `BOOLEAN`	–	Export into a ROLAP snowflake.
`value_aggregation`	measure `VARCHAR2` D_1 m-object name `VARCHAR2` … D_n m-object name `VARCHAR2`	`ANYDATA`	Get the aggregated value of a given measure.
`fact_extraction`	table name `VARCHAR2` measure list `names_tty` D_1 m-object name `VARCHAR2` … D_n m-object name `VARCHAR2`	–	Get the aggregated values of the specified measures aggregated at the given coordinate.

Table 3.6: Query functions and procedures of `queryview_*_ty`

Listing 3.22: Using query views

```
1  DECLARE
2    sales_cube mcube_sales_cube_ty;
3
4    query_view queryview_sales_cube_ty;
5  BEGIN
6    (...)
7
8    /* obtain a new query view of the sales cube */
9    query_view := sales_cube.new_queryview;
10
11   /* add a dice expression to the query view */
12   query_view :=
13     query_view.dice('Car', 'Year2010', 'Switzerland');
14
15   /* include only measures 'revenue' and 'qtySold' */
16   query_view :=
17     query_view.project(names_tty('revenue', 'qtySold'));
18
19   /* evaluate the expression list */
20   query_view := query_view.evaluate;
21
22   /* export as heterogeneous star */
23   query_view.create_star('$star', FALSE);
24 END;
```

4 Implementation

While chapter 3 presents the extension's functionality from the user's point of view, chapter 4 is intended to give an insight into the internal functioning of the implementation. The extension package is presented from the programmer's point of view. The interested reader learns how the data are organized in tables and how Oracle's object-relational features are used to implement the extension package's functionality. The concept for representing m-objects and m-relationships using Oracle's object-relational features and meta-programming capabilities as well as the concept for the logical structure have been originally developed in [Neu10].

Dimensions, m-objects, m-cubes, etc., are represented using object types. A multitude of object types is defined to represent heterogeneities in data warehouses. Furthermore, the extension package makes heavy use of dynamic data definition and manipulation. In section 4.1, an overview is given of the different object types and their concretizations as well as a motivation for dynamically creating object types and triggers instead of implementing a generic solution. The focus lies on the interdependencies between different object types and how dynamically created object types fit in the extension package's object type system.

A closer examination of the logical structure of hetero-homogeneous dimension hierarchies and m-cubes is presented in section 4.2. The logical structure of m-cubes and hetero-homogeneous dimensions is unlike any traditional ROLAP schema. A traditional ROLAP data organization, e.g., star or snowflake schema, is insufficient when applied on heterogeneous data. Consequently, a different approach has been conceived in order to store dimensional and non-dimensional data within the Oracle database. The logical structure of hetero-homogeneous dimensions and m-cubes is based on object-relational tables.

Remarks on implementation details are presented in section 4.3; the section takes a closer look on how the extension package processes commands issued by the user. It follows the course of chapter 3. Notes on the internal realization of the main functionality are included in this section. The focus lies on the algorithmic realization of the main functionality, tricky aspects of the implementation, and how user calls to the programming interface are propagated to auxiliary packages which are subsequently presented in section 4.4.

The chapter closes with a brief explanation of error handling in section 4.5. Consistency errors are reported to the user by throwing self-defined ORA-codes. The extension package makes use of an Oracle database feature that lets the developer define customized messages for application specific error cases.

4.1 Object type system

The hetero-homogeneous hierarchies and m-cubes are represented using the object-relational features of the Oracle database. The concepts of m-objects, dimensions, m-cubes, and m-relationships are represented by object types. Generally, non instantiable object types exist for the entities used to represent the components of the m-cube-based data warehouse. These abstract super types are subsequently concretized, one concretization for each dimension and m-cube, respectively. The concretizations are created on the fly along with the creation of a specific m-cube or dimension.

This section emphasizes the relationships between the object types. The functions and procedures of the object types are not explained in detail and their depiction in diagrams is generally omitted. The internal functionality and an overview of the object type methods is given in section 4.3. It is, however, explained when a method's signature is dynamically determined, i.e., the number and/or the type of the parameters depend on an m-cube or dimension, respectively.

4.1.1 Dynamic object type creation

An important aspect of the implementation is its reliance on the dynamic SQL capabilities of PL/SQL. The implementation uses the `dbms_sql` package provided by the Oracle database and native dynamic SQL through the `EXECUTE IMMEDIATE` statement in order to dynamically create object types. Whereas the employed approach dynamically customizes the number of attributes of the object types as well as the arguments of methods, a generic implementation would rely on nested tables to reflect varying numbers of dimensions. This section explores the advantages of the dynamic type creation approach over a generic implementation.

Instead of opting for a generic solution relying on collections to reflect varying numbers of dimensions, the functionality has been implemented using the dynamic SQL capabilities provided by the Oracle database management system. A generic solution is likely to be more resource intensive in production use. While the dynamic creation of object types causes some necessary effort at the time of the definition of the m-cube's structure, subsequent usage of these dynamically created types is likely to be more efficient than the reliance on a generic implementation.

In the current version, attributes and member functions of m-cubes, m-relationships, and auxiliary object types are dynamically determined at creation time. For instance, a three-dimensional m-cube has three attributes with references to the dimension objects. M-relationships in this three-dimensional m-cube always connect three m-objects, one out of each of the m-cube's dimensions. Consequently, the object type that represents m-relationships of this m-cube possesses three reference attributes.

The conceptual approach of m-cubes could as well be realized using collections of generic object types. For example, every m-cube might store references to its dimension objects within a nested table attribute. This generic approach, however, has some drawbacks when compared to the dynamic implementation. Likewise, the dynamic creation approach has some disadvantages as well. However, the disadvantages of the dynamic

creation approach are more than outweighed by its advantages. Notable disadvantages of the dynamic creation approach are an increased code complexity of the packages that dynamically create the final object types – not to be confused with the actually created object types – and an increased run-time when creating m-cubes, m-relationships, etc., since the source code of the object types is dynamically generated. However, there are many advantages over the generic approach that back up the decision to employ a dynamic creation of object types. The advantages are explained in this chapter.

The dynamic creation of concretized subtypes with a variable number of attributes is more accurate from a semantic point of view. The conceptual model of m-cubes is more accurately reflected in the logical model when object types are created dynamically. Consequently, the end-user's task of logically implementing the conceptual data warehouse model is facilitated. An m-cube's structure is reflected in its attributes when the representing object type is dynamically adapted to the number of dimensions. For example, a three-dimensional m-cube stores the references to its dimensions in three separate attributes rather than within one nested table. Thus, the user is able to determine the m-cube's structure at a single glance. Furthermore, instead of passing a list of arguments, the method signatures exactly reflect an m-cube's actual structure. Methods are dynamically adapted to the number of dimensions. The user learns the number of arguments expected by a particular method by looking at its signature. In the generic solution, it is not possible for the user to learn, for example, whether a procedure expects references to three or four m-objects by simply looking at the method's signature. The generic approach is thus more error-prone and cumbersome for the end-user.

Large parts of error checking are taken over by the database system when object types are dynamically created. If the generic solution was employed, the extension package would have to provide means for checking type violations in the end-user's calls. If the object types, however, are created using dynamic SQL, the implementation can take advantage of the integrated type checking mechanism provided by the database management system. In the current version, object types are concretized with respect to the number of attributes, their member functions and procedures with respect to the number of parameters. Thus, it is not possible for the user to assign values to attributes and to pass arguments to member methods without taking into account the expected types or number of arguments. Such errors can be checked by the database management system at compile-time. The extension package is spared from taking over this task and run-time complexity is reduced as user commands are rejected at compile-time, before their actual execution.

Consider function `create_mrel` of m-cubes as an example for concretization. Bear in mind that each dimension provides its own concretization of `mobject_ty`, namely `mobject_*_ty`. The number of parameters of function `create_mrel` now varies with the m-cube, depending on the number of the m-cube's dimensions; so do the types of these parameters. Each parameter of the `create_mrel` function represents one of the m-objects connected by the m-relationship that is to be created. These parameters demand the concrete type – not the abstract base type – and it is not possible to pass a reference to an m-object of one dimension when an m-object of another dimension is required. For example, an m-object of the *product* dimension cannot be passed when an m-object

of the *time* dimension is expected. Since each dimension defines its own object type for representing m-objects – `mobject_*_ty` – such problems can be detected at compile-time since the function is dynamically created. The database management system blocks any attempt to pass an m-object of the wrong dimension. Furthermore, if the object types are dynamically created, it is not possible to pass more m-objects to the function than the m-cube has dimensions. Again, the database system takes care of these checks at compile-time.

Another major advantage of the dynamic type creation approach is the possibility to define a custom query and data definition language over the created object types and their measures. Such a language is more easily mapped on dynamically generated object types and methods. A data definition language for m-relationships may thus be defined, mapping custom commands on the PL/SQL representation. For example, the command `CREATE MREL ('Product', 'Time', 'Location') IN MCUBE sales_cube` is easily mapped on the `create_mrel` function of the *sales* cube object.

To a certain extent, run-time performance can be ameliorated if object types are dynamically created. As already mentioned, some error checks can be performed at compile-time, thus eliminating the need for some sorts of run-time checks. On the other hand, the dynamic creation of object types initially has a negative effect on run-time. When m-cubes, m-relationships, etc., are created, a dedicated object type is dynamically created which in turn increases run-time when compared to a generic approach. However, the overhead during operation is reduced since the total number of dynamic SQL statements can be decreased when SQL statements are hard-coded into the method body during the initial creation of the object types. Furthermore, when specializing object types with respect to the number of dimensions, indexes may be defined more easily. For example, m-relationships have a variable number of attributes referencing one m-object for each of the m-cube's dimensions. In order to improve performance, an index may be defined over coordinates of m-relationships. Over a generic implementation using collections to hold references to an m-relationship's m-objects, however, an index cannot be defined. In this respect, the dynamic solution is superior to a generic approach.

A disadvantage of the dynamic type creation approach is the increased effort that is needed to change an existing schema. When changing the structure of the object types, all types need to be re-compiled and existing objects in the database are void. This is a common issue in object-oriented databases. Once defined, the schema is not easily altered, entailing the necessity to update existing data. This problem would also occur with a generic solution. To a certain extent, the Oracle database feature of *type evolution* could be used to alter existing object types, even preserving the objects that have already been defined for the altered object type [Ora09]. Since the data warehouse schema can be assumed to be rather stable this issue is not seen as an urgent problem.

4.1.2 Dimensions and m-objects

The concept of m-objects is represented by object type `mobject_ty`. This type is not instantiable (abstract) and not final, implementing most of the functionality as well as providing a common interface shared by all of its subtypes. Each dimension defines its

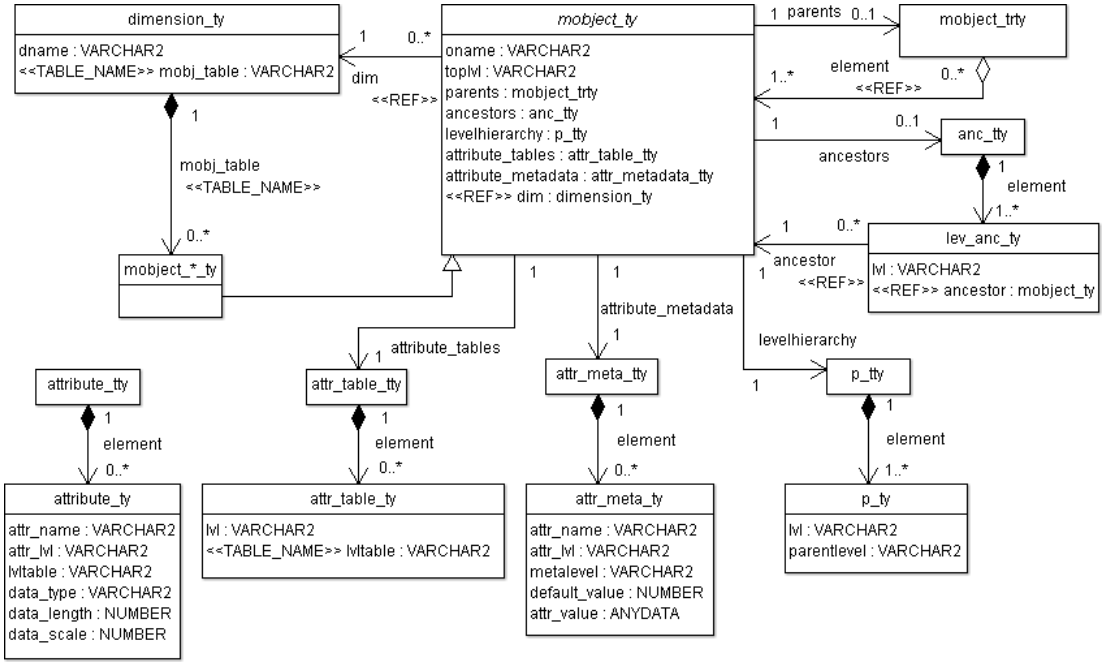


Figure 4.1: Object type representation of dimensions and m-objects

own concretization of `mobject_ty`. These specializations are generally named after the defining dimension, e.g., `mobject_product_dim_ty` for m-objects in the *product* dimension, and are denoted by `mobject_*_ty`.

The concept of dimensions is represented by object type `dimension_ty`. A dimension subsumes a set of m-objects which are stored in the dimension's m-object table. Type `dimension_ty` is not a collection in the traditional sense; it is not a table type, array, or any other PL/SQL collection type. Rather, the m-objects of a dimension are stored in an m-object table (see section 4.2). Each dimension defines its own m-object type concretization – `mobject_*_ty`. A dimension's m-object table only contains m-objects of the dimension's m-object type concretization. Figure 4.1 illustrates the relation between m-objects and dimensions. The stereotype *«TABLE_NAME»* is used to denote that `dimension_ty` is not a PL/SQL collection type; dimension attribute `mobj_table` holds the name of the table that contains the dimension's m-objects.

M-objects hold references to their parent m-objects. Attribute `parents` stores links to an m-object's parents, i.e., its direct ancestors. Table type `mobject_trty` holds a list of references to objects of type `mobject_ty`. If an m-object has no parent m-objects – i.e., a dimension's root m-object – attribute `parents` is `NULL` (see multiplicity in figure 4.1). An m-object's (direct and transitive) ancestors are contained in attribute `ancestors`. The ancestors are calculated when the m-object is created (see section 4.3) and stored within a nested table in the dimension's m-object table (see section 4.2). If an m-object has no parent m-objects, attribute `ancestors` is `NULL` as well. Attribute `ancestors` is a table type (`anc_tty`) of type `lev_anc_ty`. Object type `lev_anc_ty` holds a reference to an

ancestor m-object in attribute `ancestor`; stereotype *«REF»* in figure 4.1 is used to denote that the attribute's type is a reference. Furthermore, the ancestor m-object's top-level is stored by objects of type `lev_anc_ty`. Semantically, this representation has to be read as follows: For instance, m-object *DaVinciCode* is a *Book* when regarded from the *category* level, and a *Product* when regarded from the *top* level. Nested table `ancestors` of m-object *DaVinciCode* consequently has two tuples: `lev_anc_ty('category', REF Book)` and `lev_anc_ty('top', REF Product)`.

Most of the functionality needed to maintain m-objects is implemented in the abstract super type `mobject_ty`. Indeed, in the case of `mobject_*_ty`, the main purpose of the dynamic concretizations is to ensure type safety when later employed in m-relationships. Methods for persisting and deleting an m-object, however, are overridden and specialized which represents a certain performance gain since no dynamic SQL is needed for these tasks after the initial creation of the object type. Note that the concretized object type `mobject_*_ty` inherits all attributes from super type `mobject_ty` and does not define any attributes on its own.

The concretized subtypes of `mobject_ty` are created by the dimension's constructor function. The dimension type, in turn, is not dynamically specialized. The need for dynamically created dimension types was considered but is not necessary. There is only one case where it could be useful, namely with the dimension references of the m-object type. However, the benefit is small, which will be clarified when the m-cube type is examined.

M-objects store a reference to the dimension they belong to. The dimension reference is retrieved through an SQL statement in the constructor function of the m-object concretization `mobject_*_ty`. This SQL `SELECT INTO` statement for retrieving the reference is written hard-coded into the constructor body of `mobject_*_ty`, i.e., the name of the m-object's dimension in the statement's `WHERE` clause is pre-determined and cannot be altered at run-time. The dimension's name is determined when the type is dynamically created in the constructor of `dimension_ty`. In most cases, storing a reference to an m-object's dimension would not be necessary. Instead, hard-coded SQL `SELECT INTO` queries as in the initial retrieval in the constructor could replace the `utl_ref.select_object` call on the dimension reference attribute whenever the dimension object is needed. The dimension's name would be hard-coded into the query. However, since the Oracle database does not allow to define order functions on subtypes (error code PLS-00646), the order function `compare_to` is implemented by the abstract super type. Thus, it is necessary to store a reference to the m-object's dimension in the abstract m-object super type. The reference is then filled by the constructor of the concretization – `mobject_*_ty` – which is possible since objects inherit attributes from their parent objects. A drawback of this implementation is that the user might alter the dimension reference, which would cause erroneous results. On the other hand, the `compare_to` function can now be used in SQL queries to order result sets.

SQL-DML statements for persisting changes made to an m-object are hard-coded into the body of the `persist` procedure. This approach demands the dynamic specialization of `mobject_ty`. It could be avoided if hard-coded SQL-DML statements were replaced by dynamic SQL statements that are executed each time the procedure is called. However,

this would be less efficient. Object type `mobject_*_ty` requires the use of dynamic SQL when it is created and spares a number of dynamic SQL calls which would be necessary if super type `mobject_ty` had all functionality implemented. It is possible to hard-code the table name into the `persist` function since exactly one table is associated with m-objects of one dimensions, i.e., a one-to-one relationship between an object type and the object tables defined upon it. The same is true for m-relationships. This will be clarified together with the logical structure in section 4.2.

M-object attributes are stored in separate tables. An m-object that introduces a new attribute creates an attribute table. The names of the attribute tables are stored within nested table `attribute_tables` in type `mobject_ty`. Table type `attr_table_tty` of object type `attr_table_ty` is used to hold the names of these tables (see figure 4.1). Stereotype *«TABLE_ NAME»* is used to indicate that attribute `lvltable` of object type `attr_table_ty` holds the name of the table. Attribute metadata, on the other hand, are stored in nested table `attribute_metadata` of object type `mobject_ty`. The logical structure of m-objects is explained in section 4.2.

M-objects do not maintain a list of attributes introduced or inherited by them. Instead, attributes are only stored within the attribute tables. In order to represent information about attributes, objects of type `attribute_ty` are created. Object type `attribute_ty` defines attributes to hold information about an m-object attribute, i.e., the table where the values are stored, and information about the m-object attribute's data type.

4.1.3 M-cubes and query views

Abstract type `mcube_ty` defines only two attributes and one member procedure. Only the m-cube's name (`cname`) and the name of its m-relationship table (`mrel_table`) as well as the `delete_mcube` procedure are shared by all m-cubes. Type `mcube_ty` merely provides interface inheritance so that concretized m-cube types can be stored in the same table and returned by functions. No type body is defined for `mcube_ty`. All of the m-cube's functionality is implemented in the m-cube object type concretizations – `mcube_*_ty`.

Most importantly, the number of dimensions is hard-coded within the m-cube. A three-dimensional m-cube consequently has three attributes that hold references to its dimensions. The number of dimensions cannot be altered once the m-cube has been created. This makes the implementation more robust to user errors in the sense that a change in the number of dimensions would result in the necessity of a tail of other entities, e.g., m-relationships, to being updated as well. This is prevented by setting the number of dimensions at the time of the m-cube's creation. However, the schema migration problem commonly associated with object-oriented databases is not resolved. If the schema needs to be altered, e.g., introducing an additional or altering an existing dimension, the object types need to be re-compiled and all object types have to be re-created. Future version of the extension package may include functionality to dynamically alter existing object types. The Oracle *type evolution* feature could then be used to implement this functionality.

Along with each m-cube specialization, m-relationship types and table types defined upon them are created. These m-relationship types require some specialized types de-

pendent on the m-cube to be created for them as well. For instance, a coordinate is strictly cube-specific; as is a measure's connection level. The required types are consequently created together with the m-cube type concretization. These type dependencies are depicted in figure 4.2.

The signatures of some functions and procedures are dynamically determined, i.e., the number and/or the types of the method's parameters are determined at the time of the m-cube's creation. The parameter list of these functions and procedures depends on the dimensions referenced by the m-cube. Table 4.1 contains the m-cube methods whose signatures are dynamically determined and vary from cube to cube, where $D_1 \dots D_n$ denote the names of the m-cube's dimensions and C denotes the m-cube's name. For example, a dimension's specific m-object type is thus denoted by `mobject_`D_n`_ty`, the m-cube's specific m-cube type by `mcube_`C`_ty`. The constructor function as well as member functions `create_mrel`, `dice`, and `slice` have parameter lists that depend on the m-cube's number of dimensions. Only parameters that change with the m-cube are included; the table omits the parameter for the cube's name that is required by the query functions. Notice that `create_mrel` takes only argument m-objects of the concrete type `REF mobject_*_ty` as coordinates in the respective dimensions. For example, to create an m-relationship in the *sales* cube, references to m-objects of types `mobject_product_dim_ty`, `mobject_time_dim_ty`, and `mobject_location_dim_ty` are passed in a strictly defined order, as determined by the signature. Thus, type safety is ensured by the database system. Furthermore, the return type of some functions is dynamically determined. For instance, function `get_root_mrel` returns an object of the concretized m-relationship object type `mrel_*_ty`.

Method	Dynamic parameters	Return
constructor	D_1 `REF dimension_ty` $\cdots$ D_n `REF dimension_ty`	`SELF`
`create_mrel`	D_1`_obj REF mobject_`D_1`_ty` $\cdots$ D_n`_obj REF mobject_`D_n`_ty`	`mrel_`C`_ty`
`get_root_mrel`	–	`mrel_`C`_ty`
`get_mrelationships`	–	`mrel_`C`_tty`
`get_mrelationships_ref`	–	`mrel_`C`_trty`
`dice`	D_1`_oname VARCHAR2` $\cdots$ D_n`_oname VARCHAR2`	`mcube_ty`
`slice`	D_1`_pred slice_predicate_ty` $\cdots$ D_n`_pred slice_predicate_ty`	`mcube_ty`
`new_queryview`	–	`queryview_`C`_ty`

Table 4.1: Member methods of `mcube_*_ty` with dynamically determined signatures

A source of error still lies in the fact that type `dimension_ty` is not concretized for each dimension. The user still could change one of the m-cube's dimensions which would result in an inconsistent m-cube as the object type representing a coordinate – `coord_*_ty` – holds references to the m-objects connected by the m-cube's m-relationships. These references are of the concrete reference type `REF mobject_*_ty`. Furthermore, some of the m-cube's methods demand objects of concrete types to be passed as arguments, e.g., `create_mrel` which takes references to concretized m-objects rather than the abstract base type. However, the necessity to concretize `dimension_ty` is not urgent but might be included in future releases. Dimensions are not used in method signatures the same way m-objects are used. In this case, a trigger that prevents the m-cube's attributes from being changed at run-time would be sufficient as well. In the current version this is not yet supported.

The m-cube query operations are realized as functions of the concretized type `mcube_*_ty`. These functions are generally object-generating queries in the sense that new and independent m-cube and m-relationship objects are created. Functions `dice` and `slice` are dynamically customized. Function `dice` is adapted with respect to the m-cube's number of dimensions. The `dice` function takes for each of the m-cube's dimensions the name of an m-object which represents the new m-cube's root m-object in the respective dimension. Function `slice` takes for each of the m-cube's dimensions a predicate of type `slice_predicate_ty`. The `slice` function's parameter list is dynamically adapted to the m-cube's number of dimensions. Function `project` contains a list of measures that are to be included in the new m-cube. The `project` function's signature is not dynamically changed and could thus be defined in the abstract super type. In order to be consistent – the same cannot be done for other query operations – this has been omitted. The function's body would be implemented by the concretized type `mcube_*_ty` in any way, though.

Type `slice_predicate_ty` which is used for the `slice` function is not concretized but a generic implementation, i.e., each dimension and m-cube uses the same object type. Slice predicates are level-specific, i.e., they are always defined for a specific m-object level. A slice predicate contains a list of boolean expressions: Table type `bool_slice_expr_tty` of type `bool_slice_expr_ty` is used to represent the expression list. A boolean slice expression contains the name of the attribute, a boolean operator ($=$, $>$, $<$, etc.) and an attribute value. The expressions in the attribute list represent a conjunction of the expression elements. Function `satisfies` is used internally by the `slice` method to determine whether a given m-object satisfies the predicate. Its internal functionality is explained in section 4.3.

Dedicated *query views* are proposed for object-preserving queries. The closed m-cube query operations implemented by `mcube_*_ty` are object-generating; this is time- and space-consuming. A new m-cube is built each time a query operation is applied on the m-cube. Instead, function `new_queryview` of `mcube_*_ty` returns a query view that maintains a list of query expressions. These query expressions are not executed immediately in contrast to the query operations of `mcube_*_ty`. The concept of query views is represented by abstract base type `queryview_ty`. Every query view references its base m-cube for which the view has been defined. Each m-cube defines its own concretization

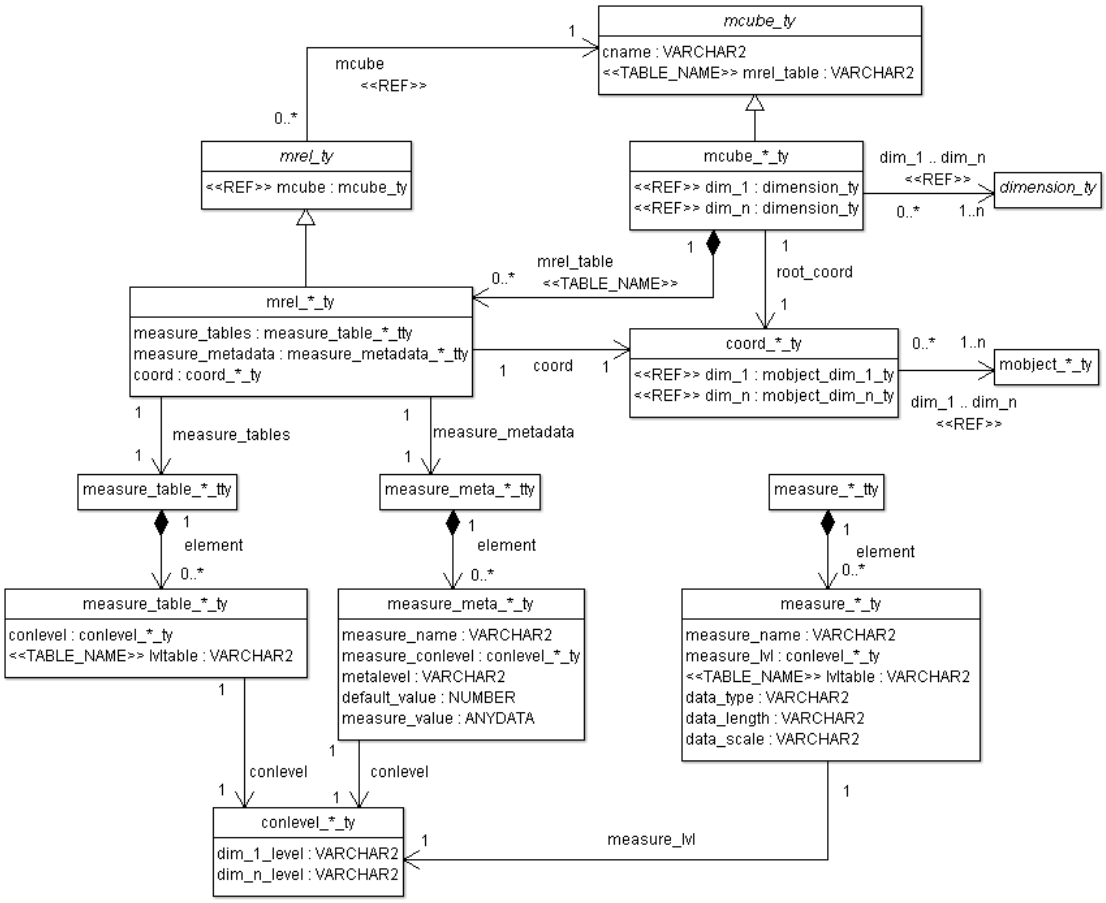


Figure 4.2: Object type representation of m-cubes and m-relationships

of this abstract base type, namely `queryview_*_ty`. The naming convention is as follows: `queryview_`⟨*name of m-cube*⟩`_ty`. The concretized type `queryview_*_ty` is returned by function `new_queryview` of m-cubes. Query views have a set of m-relationships of type `mrel_*_ty` – the m-relationship object type concretization of the query view's base m-cube – and a set of measures represented as a list of measure names. Furthermore, the query view has a root-coordinate that initially – i.e., before the evaluation of the query expressions – equals the base m-cube's root-coordinate.

A query view maintains a list of query expressions (attribute `expression_list`) of type `expr_*_tty` which is defined as a table of `expr_*_ty`. Each m-cube thus defines its own expression type. The expression type `expr_*_ty` is the abstract super type of the object types representing dice, slice and project operations. The asterisk in the type name is to be replaced by the m-cube's name. The abstract base type only defines a `descr` attribute containing a textual description of the concrete expression. The attribute is introduced for the sole reason that the Oracle database demands each object type to define at least

one attribute. The value of this attribute is always set to the name of the expression – i.e., *dice*, *slice*, and *project* – by the constructor.

A dice expression is represented by object type `dice_expr_*_ty` which is a subtype of `expr_*_ty`. A dice expression represents the new root-coordinate of an m-cube. Consequently, the number of attributes corresponds to the number of dimensions of the m-cube this dice operation is applied on. A dice expression stores references to m-objects of the m-cube's dimensions. References to the concretized m-object types are used instead of the abstract super type.

Objects of type `slice_expr_*_ty` are the cube-specific representation of slice expressions in query views. Object type `slice_expr_*_ty` is derived from `expr_*_ty`. It is specialized with respect to the number of its attributes which corresponds to the number of the m-cube's dimensions. Each attribute holds a `slice_predicate_ty` used for selecting m-objects of the respective dimension. For example, a slice expression for a three-dimensional *sales* cube may contain attributes `product_dim_pred`, `time_dim_pred`, and `location_dim_pred`, where `product_dim_pred` is a `slice_predicate_ty` that holds the selection criteria for m-objects of the *product* dimension.

Project expressions – much like the m-cube's `project` function – do not define customized signatures that depend on the m-cube. In this case, even the function and procedure bodies are not m-cube dependent. No common, m-cube independent object type for project expressions has been defined though. In order to be consistent – there are no other expressions that are not entirely m-cube dependent – the project expression object type is dynamically created as well.

A query view object provides the same query functions as `mcube_*_ty`. Functions `dice`, `slice`, and `project` of `queryview_*_ty` have almost the same interfaces as their `mcube_*_ty` counterparts. The only difference is that no name for a result m-cube is specified since no result m-cube is created. When functions `dice`, `slice`, and `project` of `queryview_*_ty` are called, instead of immediately applying the query operation on the base m-cube, a corresponding query expression object is appended to the expression list maintained by the query view.

Once the user has specified all query operations that should be applied on the base m-cube, function `evaluate` can be called. The m-relationship and measure sets are updated accordingly. The query is fully object-preserving; only the query view's m-relationship and measure sets are modified. The view can be exported into star or snowflake schema tables or facts may be extracted from the reduced set of m-relationships. Furthermore, the query view can be materialized, i.e., the m-relationship references are stored into a separate table (see chapter 3).

4.1.4 M-relationships

The abstract m-relationship object type `mrel_ty` is the super type of all the concretized m-relationship types that are defined along with the m-cubes. Only one attribute is defined by this object type, namely an attribute that holds the reference to the m-cube the m-relationship belongs to. The abstract super type is defined so that generic data structures and methods on m-relationships of different m-cubes are supported. Generally,

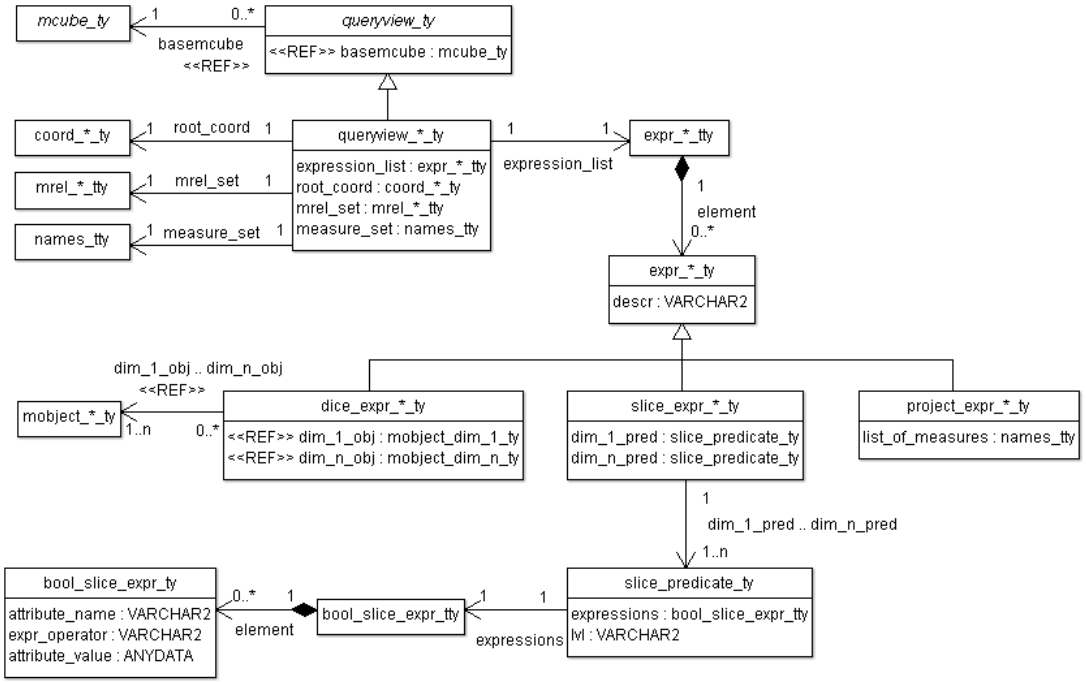


Figure 4.3: Object type representation of query views and expressions

the implementation relies concrete object types, e.g., m-cubes return the concretized subtype `mrel_*_ty` instead of the abstract super type, and the m-relationship tables of an m-cube are of concrete type `mcube_*_ty`. In one case, however, the abstract super type `mrel_ty` is used. The function in auxiliary package `consistent_mcube` that checks m-relationships for compliance with the unique induction rule for measures makes use of the abstract super type (see section 4.4).

Coordinates and connection levels are cube-specific as well. A coordinate represents a position within a cube and has as many dimensions as the cube it belongs to. There is no abstract base type defined, since it is not used for generic attributes or functions. The naming scheme is as follows: `coord_`⟨*name of m-cube*⟩`_ty`. A connection level represents the level of granularity of measures; its number of dimensions also corresponds to the m-cube's number of dimensions. There is no abstract base type for coordinates either. The naming scheme follows the usual pattern: `conlevel_`⟨*name of m-cube*⟩`_ty`.

A coordinate identifies a position within the cube. It is always cube-specific. The attributes of a coordinate object store references to m-objects of the cube. These attributes are strongly typed. For example, consider the three-dimensional *sales* m-cube with *product*, *time*, and *location* dimensions. The concretized object type `coord_sales_cube_ty` defines three attributes of types `REF mobject_product_dim_ty`, `REF mobject_time_dim_ty`, and `REF mobject_location_dim_ty`. These reference attributes are named after the respective dimension. It is not possible for the user to assign, for instance, a reference to an m-object of the *time* dimension to attribute

`product_dim` which expects a reference to an m-object of the *product* dimension (type `REF mobject_product_dim_ty`).

The sources of motivation for the utilization of a dedicated object type for coordinates instead of adding the reference attributes to the m-relationships directly are two-fold. First, coordinates are independent of the m-relationship. M-relationships are not the only database entities that make use of coordinates. M-cubes and query views also define a root-coordinate which is represented by an object of type `coord_*_ty` as well. Second, the checks for partial order, overlapping coordinates as well as equality can be incorporated in the coordinate type's member methods. By the way, the representation of an m-relationship's coordinate through the use of a separate object type is more correct from the semantics point of view.

A connection level defines a measure's level of granularity. Connection levels are cube-specific; the number of attributes depend on the number of dimensions of the m-cube. The naming scheme is as follows: `conlevel_`⟨*name of m-cube*⟩`_ty`. The attributes are of type `VARCHAR2` and contain an m-object level of the respective dimensions. An attribute's name contains the name of the dimension the level has been taken from, e.g., `product_dim_level` for a level of the *product* dimension. A connection level object type has been introduced to facilitate development. Checks for equality and level order can be incorporated as member methods. The connection levels are used in all object types dealing with measures.

Both connection level and coordinate objects are stored as column objects. There is no coordinate or connection level table. An m-relationship always stores a coordinate object, not a reference to a coordinate; the same is true for an m-cube's root-coordinate. This means that every coordinate belongs to exactly one m-relationship and m-cube, even if these objects represent the same coordinate within a particular cube. This fact is also reflected in figure 4.2. Since PL/SQL distinguishes between objects and object references – with m-cubes and m-relationships referencing objects – two m-relationships logically having the same coordinate within an m-cube physically reference two different objects.

The way measures of m-relationships are stored in the database closely resembles the way attributes of m-objects are represented. The measures are stored in measure tables (see section 4.2). An m-relationship that introduces a new measure creates a measure table. The names of the measure tables are stored within nested table `measure_tables` in type `mrel_*_ty`. This nested table (`measure_table_*_tty`) is a table of type `measure_table_*_ty`. Each measure table contains measures on the same connection level. One attribute of `measure_table_*_ty` thus contains a cube-specific connection level object (attribute `conlevel` of type `conlevel_*_ty`) and the name of the measure table (attribute `lvltable`). Metadata for measures is stored in a nested table type (`measure_meta_*_tty` of object type `measure_meta_*_ty`. It is a cube-specific object type since it contains a connection level. Figure 4.2 depicts the two types used to store measures – or actually the name of table where the values are stored – and measure metadata.

Type `measure_*_ty` is used to hold information about measures (see figure 4.2). It contains a measure's name, connection level, the name of the table where the values

are stored and information about the data type. The type is used to return information about a measure by function `has_measure` and `list_measures` of type `mrel_*_ty`. Type `measure_*_ty` is a cube-specific object type since the connection level representation (cube-specific type `conlevel_*_ty`) is referenced.

4.2 Logical structure

Traditional ROLAP approaches are insufficient for representing heterogeneities in data warehouses. Representing heterogeneities in existing ROLAP data organizations, e.g., the star and snowflake schema, would entail an extensive amount of `NULL` values in dimension and fact tables as well as the introduction of dummy levels together with an increase in complexity (see section 4.2.4). A different data organization, unlike any traditional ROLAP structure, was conceived in order to represent hetero-homogeneous hierarchies and m-cubes in data warehouses. The prototypical implementation realizes hetero-homogeneous hierarchies and m-cubes within an object-relational setting. The logical structure relies both on object tables and pure relational tables.

A particularity of the implementation is its utilization of object tables and their relation to the object types they have been defined on. Object types are commonly used as the basis for a multitude of tables. They form a pattern that can be used to define tables. The object types for m-objects and m-relationships, on the other hand, are used as a pattern for exactly one object table. The relation between tables and object types in this special case is a one-to-one relationship.

4.2.1 The logical structure of dimensions

Dimensions are represented as objects and stored in the `dimensions` table. The `dimensions` table is an object table and thus its columns correspond to the attributes of object type `dimension_ty`. The primary key of the `dimensions` table is attribute `dname` which represents the unique name of a dimension. Attribute `mobj_table` contains the name of the table where the m-objects that belong to the dimension are stored. Each dimension stores the global level-hierarchy. A dimension's global level-hierarchy is updated each time a new m-object is added to the dimension. This is ensured by dynamically created triggers.

When function `create_dimension` of package `mcube` is called to create a dimension, a new object of type `dimension_ty` is created and subsequently inserted into the `dimensions` table. Table 4.2 illustrates the resulting entries in the `dimensions` table after the *product*, *time*, and *location* dimensions have been created. Note that immediately after its creation, a dimension's level-hierarchy is empty since there have not yet been any m-objects added to the dimension.

Storing a dimension's level-hierarchy within a separate attribute is redundant information. A dimension's level-hierarchy can be deduced from the level-hierarchies of the dimension's m-objects. For the sake of efficiency, the dimension's global level-hierarchy is stored separately within nested table `levelhierarchy` of type `dimension_ty`. The

DNAME	MOBJ_TABLE	LEVELHIERARCHY	
		LVL	PARENTLEVEL
product_dim	product_dim		
time_dim	time_dim		
location_dim	location_dim		

Table 4.2: Table `dimensions` after the creation of dimensions *product*, *time*, and *location*

dimension's level-hierarchy is updated each time a new m-object is added to the dimension. Dynamically created triggers take over the task of keeping the nested table consistent with the dimension's actual level-hierarchy which would be obtained when iterating through the dimension's m-objects. The triggers are created together with the new dimension by function `create_dimension` of package `mcube`.

After adding m-objects to the dimensions of the *sales* m-cube, namely the *product*, *time*, and *location* dimension, nested table `levelhierarchy` in the `dimensions` table contains the global level-hierarchy of the dimensions. Note that a global order of levels within a dimension follows from the unique induction rule for levels. For example, while the *product* dimensions root m-object only defines levels *top*, *category* and *model*, m-object *Car* with top-level *category* introduces the additional level *brand* under level *category* as parent of level *model*. The dimension's global level-hierarchy thus consists of these four levels. In the dimension's global level-hierarchy, level *model* is a sub-level of *brand* whereas the levelhierarchy of m-object *Book* does not contain level *brand* at all and instead defines attribute *category* as the parent-level of *model*. The relative order, however, is consistent.

Table 4.3 shows the contents of the `dimensions` table after the m-objects have been added to their respective dimension. For the *time* dimension, the dimension's level-hierarchy corresponds to the root m-object's level-hierarchy since no ancestor m-object introduces an additional level. However, the level-hierarchy of the root m-object of the *product* dimension – m-object *Product* – differs from the dimension's global level-hierarchy as m-object *Car* introduces an additional level.

4.2.2 The logical structure of m-objects

M-objects are stored in object tables. Each dimension defines its own m-object table. The name of the table basically corresponds to the dimension's unique name and is explicitly stored in a column within the `dimensions` table. The primary key of each m-object table is the m-object's name, which is unique within one dimension. The columns of the m-object tables correspond to the attributes of object type `mobject_ty`.

Adding m-objects to dimensions

Each m-object table is an object table of the dimension's concretized m-object object type (`mobject_*_ty`). Thus, the functions and procedures of `mobject_*_ty` and its parent object type can be accessed after retrieving the value through an SQL `SELECT`

DNAME	MOBJ_TABLE	LEVELHIERARCHY	
		LVL	PARENTLEVEL
product_dim	product_dim	top	NULL
		category	top
		brand	category
		model	brand
time_dim	time_dim	top	NULL
		year	top
		month	year
location_dim	location_dim	top	NULL
		country	top
		region	top
		kanton	country
		city	region
		city	kanton
		store	city

Table 4.3: Table `dimensions` after adding m-objects to the dimensions

statement. M-objects store their unique name and their top-level as well as a reference to the dimension object as flat attributes within the dimension's m-object table. Other information is stored in nested tables. M-object attribute values are stored in separate tables.

M-objects are usually derived of one or more parent m-objects. References to an m-object's parents are stored as a nested table within a dimension's m-object table. The cell in the corresponding column of the m-object table is `NULL` for the dimension's root m-object. In addition to the direct parent m-objects, each m-object stores references to all of its (transitive) ancestors. Within the `ancestors` nested table, references to the ancestor m-objects as well as the respective ancestor's top-level are stored. This allows to obtain information about an m-object's class at a particular level. For instance, m-object *FiatPunto55* at level *model* is a *Car* when looked at from level *category*. Again, this is redundant information which could also be obtained by iterating through the dimension hierarchy, following the links in the nested table storing the references to the direct parent m-objects. However, the information is calculated only once on the m-object's creation. For the sake of improved performance, it can subsequently be obtained from nested table *ancestors* of object type `anc_tty`.

Table 4.4 illustrates the m-object table of the *product* dimension known from previous examples. The dimension's root m-object is *Product* for it has no parents. The `parents` nested table is `NULL` for m-objects with no parent m-objects; so is the `ancestors` nested table. M-object *Product* defines three levels: *top* $\succ$ *category* $\succ$ *model*. M-object *Book* is a direct ancestor of m-object *Product*. In the `parents` nested table, only the reference to m-object *Product* is contained. Nested table `ancestors` is calculated automatically. The entries in the `ancestors` nested table signify that on level *top*, m-object *Book* is a

ONAME	TOPLVL	PARENTS	ANCESTORS		LEVELHIERARCHY	
			LVL	ANCESTOR	LVL	PARENT
Product	top	NULL	NULL		top	NULL
					category	top
					model	category
Car	category	REF Product	top	REF Product	category	top
					brand	category
					model	brand
FiatPunto	brand	REF Car	top	REF Product	brand	category
			category	REF Car	model	brand
FiatPunto55	model	REF FiatPunto	top	REF Product	model	brand
			category	REF Car		
			brand	REF FiatPunto		
Book	category	REF Product	top	REF Product	category	top
					model	category
DaVinciCode	model	REF Product	top	REF Product	model	category
			category	REF Book		

Table 4.4: M-object table of the *product* dimension after inserting some m-objects

ONAME	TOPLVL	PARENTS	ANCESTORS		LEVELHIERARCHY	
			LVL	ANCESTOR	LVL	PARENT
Location	top	NULL	NULL		top	NULL
					country	top
					region	top
					city	country
					city	region
Austria	country	REF Location	top	REF Location	country	top
					city	country
Alps	region	REF Location	top	REF Location	region	top
					city	region
Salzburg	city	REF Austria	top	REF Location	city	country
		REF Alps	country	REF Austria	city	region
			region	REF Alps		

Table 4.5: M-object table of the *location* dimension after inserting some m-objects

Product. The level-hierarchy of *Book* is given from level *category* – the m-object's top-level – downwards. M-object *Car* in turn defines an additional level *brand* underneath level *category* as the parent-level of *model*. When m-object *Car* is created, the additional level is inserted into the dimension's global level-hierarchy.

Table 4.5 illustrates the m-object table of the *location* dimension after some m-objects have been introduced. The *location* dimension's root m-object is *Location*. The root m-object defines the following level-hierarchy: *top* $\succ$ *country* $\succ$ *city*, *top* $\succ$ *region* $\succ$ *city*. The *location* dimension presents an exemplary case of alternative aggregation paths. Level *city* is a sub-level of both level *country* and *region*. Table 4.5 illustrates how alternative aggregation paths are represented within m-object tables. The level-hierarchy contains two entries for level *city*, thus depicting alternative aggregation paths: `p_ty('city','country')` and `p_ty('city','region')`. Furthermore, each m-object with top-level *city* has two parent m-objects: one m-object with top-level *country*, another m-object with top-level *region*. Nested table `parents` of m-object *Salzburg* with top-level *city* therefore contains two tuples with references to m-object *Austria* with top-level *country* and m-object *Alps* with top-level *region*. The alternative aggregation path is also reflected in the `ancestors` nested table.

Storing attributes of m-objects

Attribute values of m-objects are stored in separate tables. Attributes are thus not stored in a dimension's m-object table. Rather, m-objects create separate tables to hold the attribute values. M-object attributes are always defined for a specific level. These attributes are inherited by descendant m-objects. M-objects assert values only for attributes at their top-level. Thus, attributes are grouped by levels; a fact reflected by the logical structure. An attribute table always contains attribute values of a particular level. All attributes within the same attribute table share the same level.

The logical structure of storing m-object attributes within separate tables was conceived in order to provide the end-user with a familiar form of data organization. The thus obtained attribute tables resemble the dimension tables of traditional ROLAP schemata – specifically the snowflake schema. The relational query semantics used to extract information from the attribute tables is more familiar to the end-user acquainted with ROLAP data warehouses and relational tables. Furthermore, well-known relational queries and algorithms designed to query ROLAP data warehouses can more easily be re-used in hetero-homogeneous data warehouses. This facilitates the integration into an existing data warehouse infrastructure.

Attribute tables are dynamically created when new attributes are added to an m-object. All attribute values within the same attribute table belong to the same level. Moreover, all attributes within the same table have been originally introduced by the same m-object. Thus, there might well exist different attribute tables for attributes defined at the same level. An m-object therefore might store its attribute values – even if they are at the same level – in different attribute tables, depending on the m-object that originally introduced the table. Furthermore, m-objects usually assert values for attributes introduced by one of their ancestor m-objects and consequently write the

values to tables not created by themself. For example, attribute *costs* at level *model* might be introduced by m-object *Product*. Values for attributes at level *model* introduced by m-object *Product* are stored within the same table – `product_dim_product_model`. However, m-object *Car* under m-object *Product* might define an additional attribute *maxSpeed* at level *model*. Values for attributes at level *model* introduced by m-object *Car* are stored in a different table – `product_dim_car_model`. Consequently, when values for attribute *costs* and *maxSpeed* are given by m-object *FiatPunto55* at level *model*, these values are written to different tables.

From another point of view, it can be said that m-objects write attribute values only to tables of whom they inherit (or introduce) all the attributes that are contained. M-object *DaVinciCode* at top-level *model* which is a direct ancestor of *Book* does not inherit attribute *maxSpeed* and only gives a value for attribute *costs*. M-object *DaVinciCode* consequently writes values only to table `product_dim_product_model`. Within a hetero-homogeneous data warehouse, the attribute tables are homogeneous. Each attribute table is homogeneous with respect to the level of the contained m-object attributes. Furthermore, an attribute table does not contain `NULL` values since each m-object inserting a tuple into the table asserts a value for all attributes contained within the particular attribute table.

Another reason for choosing to store attribute values in separate tables is a performance issue. A conceivable solution would be the storage of attribute values within a nested table of a dimension's m-objects table. However, this would result in large nested tables with a multitude of tuples. Searching these tables is costly. By splitting the attribute values on various tables, thus grouping the data by level, the number of tuples to be searched can be constrained. Furthermore, attributes of the same level and introduced by the same m-object are already grouped. This could be an advantage when querying data.

The names of the attribute tables follow a specific convention: ⟨*name of dimension*⟩_⟨*name of introducing m-object*⟩_⟨*level*⟩. The attribute table names are not only given by convention. They are explicitly stored within the dimension's m-object table. A nested table of type `p_tty` is used to store the attribute tables for attributes introduced by a particular m-object on a particular level. Table type `attr_table_tty` is a table of type `attr_table_ty` which contains two attributes. As already mentioned, attribute tables contain attributes that have all been introduced by the same m-object on the same level. Thus, the first object attribute of type `attr_table_ty` holds the level of the m-object attribute table (object attribute `lvl` of type `VARCHAR2`). The second attribute of `attr_table_ty` holds the name of the attribute table (object attribute `lvltable`).

When a new attribute is introduced at a particular m-object by adding it through procedure `add_attribute` of `mobject_ty`, two possible cases have to be distinguished. First, if there does not exist an entry in the m-object's nested table `attribute_tables` where the value of object attribute `lvl` is equal to the level of the newly introduced attribute, create an attribute table and add an entry to nested table `attribute_tables` containing the level and the new table's name. Otherwise, if in the m-object's nested table `attribute_tables` already exists an entry where the value of object attribute `lvl` is equal to the level of the newly introduced attribute, no attribute table is created and

ONAME	CATMGR	TAXRATE
Car	Mr. Smith	20
Book	Mrs. Jones	10

Table 4.6: `product_category`

ONAME	COSTS
FiatPunto55	15000
DaVinciCode	10

Table 4.7: `product_model`

ONAME	MAXSPEED
FiatPunto55	160

Table 4.8: `car_model`

the asserted values will be inserted into the existing table. The existing table, however, has to be dynamically altered; a new column with the attribute's name and data type is added.

The columns containing the m-object's attribute values have the type which the user passed to procedure `add_attribute` of `mobject_ty` when the attribute is added to the m-object. Column `oname` is the primary key of an attribute table. It further references the column `oname` in the dimension's m-object table. For example, a tuple within an attribute table for m-object *FiatPunto55* must have a corresponding tuple in the m-object table with the same name. The entry in the attribute table is deleted when the m-object is removed from a dimension's m-object table.

An example will clarify the procedure. Consider the *product* dimension's root m-object *Product*. It defines attribute *catMgr* at level *category* and attributes *costs* and *taxRate* at level *model*. When adding these two attributes to the m-object by using the `add_attribute` procedure, two attribute tables will be created. The first table – `product_dim_product_category` – will hold values for attributes introduced by m-object *Product* at level *category*. The other table – `product_dim_product_model` – will hold values for attributes introduced by m-object *Product* at level *model*. M-object *Car* defines a new attribute *maxSpeed* at level *model* and gives a value for attribute *catMgr* at level *category* which it inherits from its parent m-object *Product*. For attribute *maxSpeed*, a new table – `product_dim_car_model` – is created, which holds values for all attributes at level *model* introduced by m-object *Car*. M-objects *Car* and *Book* store values for *catMgr* in table `product_dim_product_category` (table 4.6). M-objects *FiatPunto55* and *DaVinciCode* give values for attribute *costs* which are written to table `product_dim_product_model` (table 4.7). At last, m-object *FiatPunto55* with top-level *model* as descendant of *Car* gives a value for attribute *maxSpeed* and writes it to table `product_dim_car_model` (table 4.8).

Immediately after the creation of the attribute tables, nested table `attribute_tables` in the dimension's m-object table is updated. Table 4.9 shows the updated m-object table, containing those m-objects that created new attribute tables. Following the previous example, m-objects *Product* and *Car* are shown as these two m-objects introduce new attributes and consequently create new attribute tables. M-object *Product* has two

entries in the `attribute_tables` nested table, since it introduces attributes at two different levels: *category* and *model*. M-object *Car* introduces an additional attribute at level *model* – attribute *maxSpeed* – and thus creates a separate attribute table for attributes introduced at level *model* by m-object *Car*. The name of this table is stored in nested table `attribute_tables`.

ONAME	TOPLVL	ATTRIBUTE_TABLES	
		LVL	LVLTABLE
Product	top	category	product_dim_product_category
		model	product_dim_product_model
Car	category	model	product_dim_car_model

Table 4.9: Nested table `attribute_tables` of the *product* dimension's m-object table

Storing metadata about attributes

Metadata about m-objects is stored in nested tables within a dimension's m-object table. Unlike the attributes' asserted values, attribute metadata are not stored in separate tables. Table type `attr_meta_tty` stores entries of type `attr_meta_ty` which eventually hold the metadata. Object type `attr_meta_ty` has several object type attributes. The attribute's name and level are indicated by object attributes `attr_name` and `attr_lvl`. Metadata is always set at a particular meta level (attribute `metalevel`) and can either be a shared – i.e., cannot be altered by descendant m-objects – or a default value which can be altered by descendant m-objects (attribute `default_value`). The actual meta-value is of type `ANYDATA` and can store data of an arbitrary type, even references to m-objects (see chapter 3).

Table 4.10 illustrates how m-object metadata might be represented within the m-object's nested table `attribute_metadata`. Attribute *costs* introduced by m-object *Product* at level *model* has meta-value at meta level *quantity* set to value *currency*. This is a shared meta-value and cannot be altered by descendant m-objects. By default, m-object *Product* defines the *unit of measurement* of attribute *costs* to be €. This value may be altered by descendant m-objects, e.g., m-object *DodgeViper* re-defines the *unit of measurement* of attribute *costs* as *$*. From the semantics point of view, this means that the costs of all cars of brand *DodgeViper* are measured in U.S. Dollars since it is an American brand. The same thing may be done for attribute *maxSpeed*. While a car's maximum speed is generally measured in *km/h*, m-object *DodgeViper* – representing an American car brand – could alter the *unit of measurement* to *mph*.

4.2.3 The logical structure of m-cubes and m-relationships

M-cubes are stored in the `mcubes` table. The `mcubes` table is an object table of abstract type `mcube_ty` which is the super type of all concretized m-cube object types (`mcube_*_ty`). Abstract type `mcube_ty` only defines two attributes, namely the m-cube's

ONAME	ATTRIBUTE_METADATA				
	ATTR_NAME	ATTR_LVL	METALEVEL	DEFAULT	ATTR_VALUE
Product	costs	model	quantity	0	currency
	costs	model	unit	1	€
	taxRate	model	unit	0	%
Car	maxSpeed	model	unit	1	km/h
DodgeViper	costs	model	unit	0	$
	maxSpeed	model	unit	0	mph

Table 4.10: Nested table `attribute_metadata` of the *product* dimension's m-object table

unique name `cname` and the name of the m-cubes m-relationship table (`mrel_table`). Thus, the `mcubes` table contains only two attributes. However, the m-cube object can be retrieved from the object table and its member methods can be accessed.

Each m-cube defines its own m-relationship table. The m-relationship table holds all m-relationships that belong to a particular m-cube. The table's name is stored in a separate attribute in the `mcubes` table and generally equals the name of the m-cube. The m-relationship table is an object table of the concretized m-relationship object type (`mrel_*_ty`).

The way m-relationships are organized in tables resembles the logical data structure m-objects. Each m-cube has its own m-relationship table. M-relationship tables are object tables of the m-cube's concretized m-relationship type. The m-relationship table's columns thus correspond to the attributes of `mcube_*_ty`. Measures are not stored in the m-relationships directly. The measure values are stored in measure tables similar to the m-object attributes. The m-relationship table holds measure metadata and contains the names of the measure tables.

Adding m-relationships to m-cubes

Each m-cube creates its own concretized m-relationship object type – `mrel_*_ty`. When an m-relationship is created for a particular m-cube, it is inserted into the m-cube's m-relationship table. This table is an object table of type `mrel_*_ty` and only holds m-relationships of the same m-cube.

M-relationships are identified by their coordinate. Unfortunately, it is not possible to define unique or primary key constraints on `REF` columns. The primary key is thus not checked by the database management system. Nevertheless, coordinates on unambiguously identify m-relationships. In fact, m-relationships do not define any other attribute that is suitable for identification. The only other attributes contained in the m-relationship table are nested tables for storing measures and measure metadata. The coordinate is an object type whose attributes are dynamically adapted. The coordinate is no nested table; it is no collection type. The number of dimensions is hard-coded into the type definition. The `coord` attribute of the m-relationship table is divided into

separate columns. The number of these separate columns depends on the number of the m-cube's dimensions and changes from cube to cube.

An example shows how the m-relationships are represented within tables. Consider the *sales* m-cube that involves sales in Austria and Switzerland. Table 4.11 depicts the *product* dimension's m-relationship table after the insertion of the root m-relationship as well as m-relationships representing sales in *Austria* and *Switzerland*. Note that each row represents an object of type `mrel_sales_cube_ty` and could be retrieved issuing an SQL `SELECT` statement in order to have access to the object's functions and procedures. The table holds also a reference to the m-cube but is omitted in the figure; so are the columns for measures. The first row is an m-relationship defined at the m-cube's root-coordinate. There are m-relationships defined on the most specific level of granularity in their respective branch, e.g., $\langle model, month, city \rangle$ generally is the most specific connection level. However, m-object *Switzerland* defines an additional level *store* under *city* and thus a more specific connection level exists. It is not necessary for an m-cube to always define m-relationships on the most specific connection level. In this example, car sales in Switzerland are given at the most specific connection level. On the other hand, book sales for Switzerland are given on a less detailed level since the root m-object of the *location* dimension holds references to cities. For which connection levels an m-relationship is given depends on the measures and their connection level (or level of granularity) identified during the conceptual modeling.

An advantage of the dynamic creation approach over a generic implementation that uses collections to reference an m-relationship's m-objects is the possibility to define indexes on coordinates. The current version of the prototype does not take advantage of this possibility. However, future releases may exploit the potentials of indexes in terms of performance improvement. Indexing techniques will have to be evaluated regarding their suitability in the context of m-relationships. Some issues still need to be resolved before indexes can be defined. For instance, the Oracle database does not allow an index to be defined on object types (ORA-02327). Therefore, indexes might have to be implemented manually.

Storing measures in m-relationships

The way measures are stored closely resembles the logical structure of m-object attributes. Similar to m-objects storing attribute values, m-relationships store the measure values in separate, relational tables. Each measure table contains measure values at the same aggregation level, introduced by the same m-relationship. The motivation for this approach stems from a facilitated integration due to less migration and retraining effort that is necessary to introduce a hetero-homogeneous data warehouse. The organization of measures within separate measure tables closely resembles the structure of the fact tables in star and snowflake schemata; measures at the same level of granularity are generally grouped together within the same table. Existing query definitions and algorithms can be more easily adapted to the heterogeneous data warehouse.

The m-relationships depicted in table 4.11 have been created to hold measure values. A multitude of m-relationships have been created in order to store measures at

COORD			MEASURE_ TABLES	MEASURE_ METADATA
PRODUCT_DIM	TIME_DIM	LOCATION_DIM		
REF Product	REF Time	REF Location	(...)	(...)
REF FiatPunto55	REF Jan2010	REF Vienna	(...)	(...)
REF FiatPunto55	REF Feb2010	REF Vienna	(...)	(...)
REF FiatPunto55	REF Jan2010	REF Salzburg	(...)	(...)
REF FiatPunto55	REF Feb2010	REF Salzburg	(...)	(...)
REF DaVinciCode	REF Jan2010	REF Vienna	(...)	(...)
REF DaVinciCode	REF Feb2010	REF Vienna	(...)	(...)
REF DaVinciCode	REF Jan2010	REF Salzburg	(...)	(...)
REF DaVinciCode	REF Feb2010	REF Salzburg	(...)	(...)
REF Car	REF Year2010	REF Switzerland	(...)	(...)
REF FiatPunto55	REF Jan2010	REF LausanneShop1	(...)	(...)
REF FiatPunto55	REF Feb2010	REF LausanneShop1	(...)	(...)
REF FiatPunto55	REF Jan2010	REF LausanneShop2	(...)	(...)
REF FiatPunto55	REF Feb2010	REF LausanneShop2	(...)	(...)
REF DaVinciCode	REF Jan2010	REF Lausanne	(...)	(...)
REF DaVinciCode	REF Feb2010	REF Lausanne	(...)	(...)

Table 4.11: The *sales* cube's m-relationship table

COORD	MEASURE_TABLES			
	CONLEVEL			LVLTABLE
	PROD_LVL	TIME_LVL	LOC_LVL	
ProdTimeLoc	model	month	city	sales_cube_prtilo_momoci_1
	category	year	country	sales_cube_prtilo_cayeco_1
Car2010CH	model	month	store	sales_cube_cayesw_momost_1
	brand	year	city	sales_cube_cayesw_bryeci_1

Table 4.12: M-relationship table of the *product* dimension focused on measures

various levels of granularity. After these m-relationships have been added to the m-cube, measures may be added. The m-relationships that introduce a measure commonly do not assert a value for this measure. Measures are always defined for a particular connection-level. Only m-relationships with a top connection-level equal to a measure's connection-level assert a value for this particular measure. For example, the m-relationship connecting m-objects *Product*, *Time*, and *Location* introduces measure revenue at connection-level $\langle model, month, city \rangle$. All descendant m-relationships inherit this measure and define a value if their top connection-level matches the measure's connection-level. The m-relationship connection m-objects *FiatPunto55*, *Jan2010*, and *Salzburg* has top connection-level $\langle model, month, city \rangle$ and thus assigns a value for measure *revenue*.

When the m-relationship between m-objects *Product*, *Time*, and *Location* has measure *revenue* added, a new measure table is created. The name of this measure table corresponds to a shortened string representation of the m-relationship's coordinate followed by a shortened representation of the connection-level of the measure. These string representations consist of the first two characters of the identifiers for the components that define the respective objects, i.e., the m-object names for coordinates and the level names for connection-levels. For example, coordinate (*Product*, *Time*, *Location*) is represented by *prtilo*, connection-level $\langle model, month, city \rangle$ is represented by *momoci*. The shortened string representations might be ambiguous; a running number is thus appended to the table name. The naming scheme of measure tables is as follows: $\langle$*name of m-cube*$\rangle$_$\langle$*shortened representation of the m-relationship's coordinate*$\rangle$_$\langle$*shortened representation of the measure's connection-level*$\rangle$_$\langle$*running number*$\rangle$. Measures of the same connection-level introduced by the same m-relationship are grouped within the same measure table. Nested table `measure_tables` stores the names of the measure tables holding the values of measures introduced by the particular m-object.

When measure *revenue* at connection-level $\langle model, month, city \rangle$ is introduced by the m-relationship between m-objects *Product*, *Time*, and *Location* measure table `sales_cube_prtilo_momoci_1` is created. The name of this table is inserted into nested table `measure_tables` of m-relationships. The connection-level of the measures whose values are stored within this table are not only given by the naming convention; nested table `measure_tables` stores the connection-level together with the table's name. Table 4.12 illustrates how the table names are stored in nested table `measure_tables`. Note that column *coord* depicts only abbreviated placeholders for the coordinate objects as shown in table 4.12. The reader may refer to table 4.12 for the correct representation of the coordinates within an m-relationship table. Column names of the `conlevel` column are abbreviated.

The connection-level of measure *revenue* is defined at a more specific level of granularity for car sales in *Switzerland* in the year 2010. The connection-level of measure *revenue* is moved to $\langle model, month, store \rangle$. A new measure table has to be created since the measure's connection-level does not correspond to the connection-level of the same measure introduced by the ancestor m-relationship connecting *Product*, *Time*, and *Location*. All m-relationships with top connection-level $\langle model, month, store \rangle$ that are descendants of the m-relationship connecting m-objects *Car*, *Year2010*, and *Switzerland* consequently insert their value for measure *revenue* into the newly created measure table

`sales_cube_cayesw_momost_1`. Car sales in *Switzerland* in the year 2010 provide the additional measure *qtySold* representing the sold quantity within the year 2010 within a particular city, i.e., at connection-level $\langle category, year, city \rangle$. Measure *cheapestOffer*, in turn, is again introduced by the m-relationship between m-objects *Product*, *Time*, and *Location*. The measure represents the cheapest offer and is defined at connection-level $\langle category, year, country \rangle$, i.e., the cheapest offer of each product category within a particular year sold within a particular country.

The measures' values are commonly not defined by the introducing m-relationship. When a measure's value is asserted using procedure `set_measure`, the value is inserted into the corresponding measure table. The table is retrieved by iterating through the m-relationship's `measure_tables` nested table as well as the nested tables of its ancestor m-relationships. Table 4.13 shows the asserted values for measure *revenue* defined at connection-level $\langle model, month, city \rangle$. All m-relationships at top connection-level $\langle model, month, city \rangle$ – except for descendants of the m-relationship between *Car*, *Year2010*, and *Switzerland* – store the values for measure *revenue* into table `prtilo_momoci_1`. Car sales in *Switzerland* in the year 2010 are available at a more detailed level of granularity and consequently stored in a different measure table, namely `cayesw_momost_1` depicted in table 4.14. Table 4.15 depicts the measure table for measures at connection-level $\langle category, year, country \rangle$ introduced by the m-relationship between m-objects *Product*, *Time*, and *Location*. Table 4.16 depicts the measure table for measures at connection-level $\langle brand, month, city \rangle$ introduced by the m-relationship between m-objects *Car*, *Year2010*, and *Switzerland*.

Each measure has its own column in the measure table. If a new measure is added at a particular m-relationship for a connection-level which already has a measure table defined by this m-relationship, the existing measure table is altered. In this case, a new column is added to the measure table. For instance, the m-relationship between m-objects *Product*, *Time*, and *Location* might define an additional measure *turnover* at connection-level $\langle model, month, city \rangle$. For this connection-level, there already exists a measure table and thus the measure table has to be altered.

Storing metadata about measures

Metadata about measures are stored analogously to the metadata about m-object attributes. A nested table within an m-cube's m-relationship table holds measure metadata. Nested table type `measure_meta_*_tty` of object type `measure_meta_*_ty` has been defined to hold metadata about measures. These types are cube-specific since objects of type `measure_meta_*_ty` contain a measure's connection-level. The connection-level object type is cube-specific; consequently, the measure metadata object type is cube-specific as well.

The `measure_metadata` nested table of type `mrel_*_ty` stores the connection-level of the measure in the `measure_conlevel` column object of type `conlevl_*_ty`. Every metadata tuple has a meta level. This meta level is stored in a `VARCHAR2` column, which allows to represent complex metadata hierarchies. Future releases might use an m-object hierarchy to represent metadata (see chapter 3). In this case, the meta level would

COORD			REVENUE
PRODUCT_DIM	TIME_DIM	LOCATION_DIM	
REF FiatPunto55	REF Jan2010	REF Salzburg	330000
REF FiatPunto55	REF Feb2010	REF Salzburg	410000
REF FiatPunto55	REF Jan2010	REF Vienna	1350000
REF FiatPunto55	REF Feb2010	REF Vienna	1200000
REF DaVinciCode	REF Jan2010	REF Salzburg	1500
REF DaVinciCode	REF Feb2010	REF Salzburg	6300
REF DaVinciCode	REF Jan2010	REF Vienna	10500
REF DaVinciCode	REF Feb2010	REF Vienna	9300
REF DaVinciCode	REF Jan2010	REF Lausanne	2500
REF DaVinciCode	REF Feb2010	REF Lausanne	4300

Table 4.13: `prtilo_momoci_1`

COORD			REVENUE
PRODUCT_DIM	TIME_DIM	LOCATION_DIM	
REF FiatPunto55	REF Jan2010	REF LausanneShop1	230000
REF FiatPunto55	REF Feb2010	REF LausanneShop1	180000
REF FiatPunto55	REF Jan2010	REF LausanneShop2	130000
REF FiatPunto55	REF Feb2010	REF LausanneShop2	95000

Table 4.14: `cayesw_momost_1`

COORD			CHEAPESTOFFER
PRODUCT_DIM	TIME_DIM	LOCATION_DIM	
REF Car	REF Year2010	REF Austria	10000
REF Car	REF Year2010	REF Switzerland	12000
REF Book	REF Year2010	REF Switzerland	4
REF Book	REF Year2010	REF Switzerland	5

Table 4.15: `prtilo_cayeco_1`

COORD			QTYSOLD
PRODUCT_DIM	TIME_DIM	LOCATION_DIM	
REF FiatPunto	REF Year2010	REF Lausanne	31

Table 4.16: `cayesw_bryeci_1`

correspond to the top-level of an m-object. A reference to this m-object would be stored in the `measure_value` column which is of type `ANYDATA`. Measure metadata – as m-object attribute metadata – can either be shared or default.

Table 4.17 illustrates how measure metadata are stored within the `measure_metadata` nested table of the *sales* m-cube. Notice that the coordinate column objects are abbreviated due to space considerations instead of representing the m-relationships' coordinates accurately as three-attribute objects. Measure *revenue* is generally measured in euros. The m-relationship between m-objects *Product*, *Time*, and *Location* (*ProdTimeLoc*) which introduced measure *revenue* specifies the *unit of measurement* for *revenue* as €. The meta-value has been marked default and can therefore be altered by descendant m-relationships. The *unit of measurement* of measure *revenue* is set to *CHR* by the m-relationship between m-objects *Car*, *Year2010*, and *Switzerland* (*Car2010CH*), thus changing the default value given by its ancestor m-relationship. Table 4.17 further illustrates the fact that the refined level of granularity for a particular *measure* is also reflected in the `measure_metadata` nested table. Measure *revenue* is available at a more detailed level of granularity for car sales in Switzerland in the year 2010.

The way a measure's aggregation function is stored is likely to change in future releases. Currently, the aggregation function has to be stored as metadata at meta level *function*. In future releases, the user might be demanded to create an m-object hierarchy to represent measure metadata for the unit of measurement as well as the measure's aggregation function. If no aggregation function is specified the extension package's methods assume the aggregation function to be *SUM*.

COORD	MEASURE_METADATA						
	MEASURE	CONLEVEL			META LEVEL	DEF	VALUE
		PROD	TIME	LOC			
ProdTimeLoc	revenue	model	month	city	unit	1	€
	revenue	model	month	city	function	0	SUM
	cheapestOffer	category	year	country	unit	1	€
	cheapestOffer	category	year	country	function	0	MIN
Car2010CH	revenue	model	month	store	unit	1	CHF
	cheapestOffer	category	year	country	unit	1	CHF
	qtySold	brand	year	city	unit	1	number

Table 4.17: Nested table `measure_metadata` of the *sales* cube's m-relationship table

4.2.4 Flat data warehouse schemata

The extension package provides functionality that facilitates the integration of data warehouses based on m-cubes into an existing data warehouse infrastructure. The transformation of m-object dimensions, m-cubes and m-relationships into star and snowflake schemata is supported by the prototypical implementation. The current implementation only allows for the export of heterogeneous ROLAP schemata. A conversion to homo-

geneous ROLAP schemata via roll-up is currently not supported. This roll-up has to be done after schema export by applying SQL queries on the ROLAP tables.

Star schema

The star schema is the most basic form of a ROLAP organization. The ROLAP table organization according to the star schema has one dimension table for each of the m-cube's dimensions. Each tuple within a dimension table stores the non-dimensional attributes of the dimension's aggregation levels. The fact table that contains the measures references the respective entries in the dimension tables. Surrogate keys are used to identify tuples within the dimension tables.

In order to represent heterogeneities with respect to the aggregation levels of measures, column `aggregation_level` is introduced for dimension tables. This approach is commonly employed for storing aggregated values in fact tables (cf. [Hah06]). However, the approach can be adopted to represent heterogeneities as well. By introducing a level column in the dimension tables it is possible to store facts with different aggregation levels within the same fact table.

Figure 4.4 illustrates the export functionality of the extension package. Consider the three-dimensional *sales* m-cube with dimensions *product*, *time*, and *location*. The exported star organization consists of three dimension tables – one for each of the source m-cube's dimensions – and one fact table. The fact table references the dimension tables whose primary keys are surrogate integer identifiers. The fact table has a composite primary key that is composed of the table's foreign key attributes. Each entry within a dimension table corresponds to a particular m-object of the respective dimension.

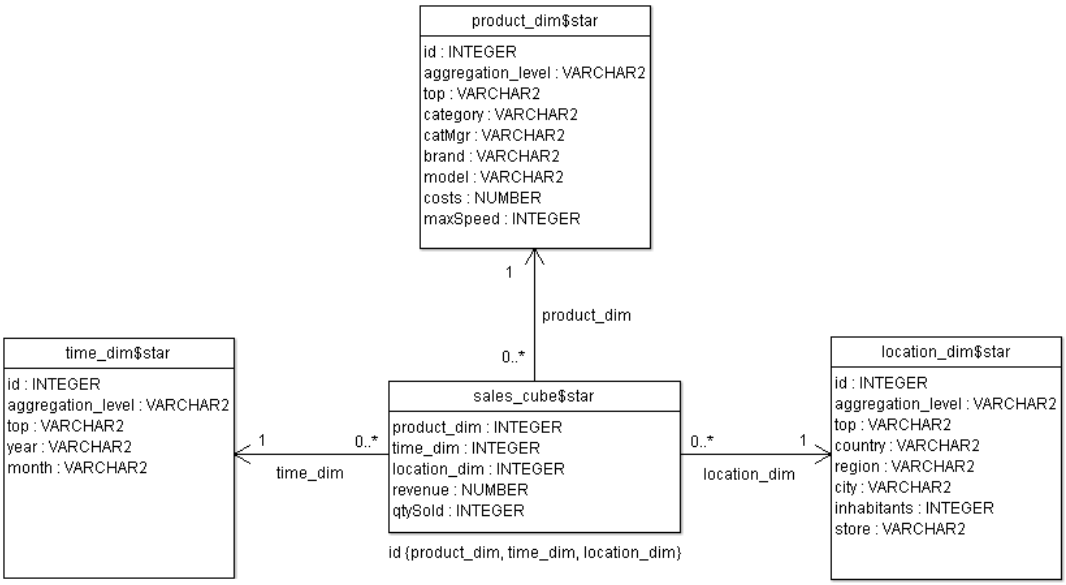


Figure 4.4: Example ROLAP organization according to the star schema

A heterogeneous star organization results in a number of `NULL` value entries – both within the fact and the dimension tables. Every m-object has a separate entry in one of the dimension tables. Column `aggregation_level` contains the respective m-object's top-level. Most of a dimension table's tuples contain `NULL` values. The `NULL` entries are a consequence of the heterogeneities in the level-hierarchies and the differences in the levels of aggregation between facts. Tables 4.18, 4.19, and 4.20 depict example dimension tables for the *sales* m-cube's three dimensions. Note that the level-hierarchy of the *location* dimension has been simplified over the example in figure 4.4 due to space considerations. The heterogeneities in the *product* dimension's level-hierarchy cause tuples with `NULL` values. For example, the entry representing m-object *DaVinciCode* at top-level *model* under m-object *Book* has a `NULL` value in column `brand`. This is due to the fact that level *brand* is introduced by m-object *Car* at top-level *category*. Consequently, all m-objects under *Book* do not have an ancestor m-object at the *brand* level and thus no value is assigned for the respective cell in the `brand` column of the *product* dimension's dimension table.

Table 4.21 shows the generated fact table of the *sales* cube. Foreign key constraints are placed on table attributes `product_dim`, `time_dim`, and `location_dim`, referencing the respective dimension table. The fact table's primary key is composed of the foreign key columns. Notice how heterogeneities are contained in the fact table. Even though measure *revenue* has been defined at varying levels of granularity, the same dimension tables are referenced for all levels of granularity. Each entry in the fact table corresponds to a particular m-relationship. The m-objects connected by these m-relationships are represented by tuples within the dimension tables. Just like m-relationships reference m-objects at different top-levels tuples in the fact table reference tuples in the dimension tables at various aggregation levels. Column `aggregation_level` of the dimension tables corresponds to the top-level of the m-object represented by the tuple within the dimension table. A tuple in the fact table storing the value of a measure defined at connection-level $\langle model, month, city \rangle$ – e.g., measure *revenue* – references (i) a tuple in the *product* dimension's dimension table with a value of *model* in column `aggregation_level`, (ii) a tuple in the *time* dimension's dimension table with a value of *month* in column `aggregation_level`, (iii) a tuple in the *location* dimension's dimension table with a value of *city* in column `aggregation_level`. Tuples in the fact table storing values of measures defined at a different connection-level reference different entries in the dimension tables – having the appropriate value in column `aggregation_level`.

Measure *revenue* is available at a more detailed level of granularity for car sales in the year 2010 in Switzerland, namely $\langle model, month, store \rangle$. Table 4.21 reflects this change in granularity since tuples representing m-relationships between m-objects *Car*, *Year2010*, and *Switzerland* reference entries in the *location* dimension's dimension table having value *store* in column `aggregation_level`.

Notice that all measures introduced within a particular m-cube are contained within only one fact table, regardless of their connection-level. Tuples in the fact table, however, can only hold values for measures at a connection-level corresponding to the aggregation level represented by the referenced entries in the dimension tables. For instance, measure *qtySold* is defined at connection-level $\langle brand, year, city \rangle$. Values for this measure

ID	AGGR_LVL	TOP	CATEGORY	BRAND	MODEL	COSTS
1	top	Product	NULL	NULL	NULL	NULL
2	category	Product	Book	NULL	NULL	NULL
3	model	Product	Book	NULL	DaVinciCode	10
4	category	Product	Car	NULL	NULL	NULL
5	brand	Product	Car	FiatPunto	NULL	NULL
6	model	Product	Car	FiatPunto	FiatPunto55	15000
7	brand	Product	Car	DodgeViper	NULL	NULL
8	model	Product	Car	DodgeViper	DodgeViperGTS	125000

Table 4.18: Dimension table of the *product* dimension according to the star schema

ID	AGGR_LVL	TOP	YEAR	MONTH
1	top	Time	NULL	NULL
2	year	Time	Year2010	NULL
3	model	Time	Year2010	Jan2010
4	model	Time	Year2010	Feb2010

Table 4.19: Dimension table of the *time* dimension according to the star schema

ID	AGGR_LVL	TOP	COUNTRY	CITY	INHAB.	STORE
1	top	Location	NULL	NULL	NULL	NULL
2	country	Location	Austria	NULL	NULL	NULL
3	city	Location	Austria	Salzburg	147685	NULL
4	country	Location	Switzerland	NULL	NULL	NULL
5	city	Location	Switzerland	Lausanne	122284	NULL
6	store	Location	Switzerland	Lausanne	122284	LausanneShop1
7	store	Location	Switzerland	Lausanne	122284	LausanneShop2

Table 4.20: Dimension table of the *location* dimension according to the star schema

cannot be stored in the same tuple as the values for measure *revenue* since the measure's connection-levels differ. This results in a great number of `NULL` values in the respective columns if the measures are rather heterogeneous with respect to their connection-levels, as can be seen in table 4.21.

PRODUCT_DIM	TIME_DIM	LOCATION_DIM	REVENUE	QTYSOLD
5 (FiatPunto)	2 (Year2010)	5 (Lausanne)	NULL	31
3 (DaVinciCode)	3 (Jan2010)	3 (Salzburg)	1500	NULL
3 (DaVinciCode)	4 (Feb2010)	3 (Salzburg)	6300	NULL
3 (DaVinciCode)	3 (Jan2010)	5 (Lausanne)	2500	NULL
3 (DaVinciCode)	4 (Feb2010)	5 (Lausanne)	4300	NULL
6 (FiatPunto55)	3 (Jan2010)	3 (Salzburg)	330000	NULL
6 (FiatPunto55)	4 (Feb2010)	3 (Salzburg)	410000	NULL
6 (FiatPunto55)	3 (Jan2010)	6 (LausanneShop1)	230000	NULL
6 (FiatPunto55)	4 (Feb2010)	7 (LausanneShop1)	180000	NULL
6 (FiatPunto55)	3 (Jan2010)	6 (LausanneShop2)	130000	NULL
6 (FiatPunto55)	4 (Feb2010)	7 (LausanneShop2)	95000	NULL

Table 4.21: Fact table of the *sales* cube according to the star schema

The export functionality creates dimension and fact tables. The star schema created by the extension package comprises ready-to-use dimension and fact tables that can be queried by the user. However, the Oracle database provides additional features for handling data warehouses. Dimension objects can be created using a `CREATE DIMENSION` statement. It is not mandatory in Oracle to create dimensions in order to use the data warehouse. However, the support for query rewriting is better when dimensions are defined [Lan09]. Future releases of the extension package may automate the definition of Oracle dimensions along with the export of the data into star schema tables. The same applies to the snowflake schema export.

No indexing techniques are employed by the extension package. Future releases may include the possibility of defining bitmap indexes on fact tables or bitmap join indexes on fact and dimension tables in order speed up the performance of OLAP queries in the created star schema [Lan09]. In the current version, the user may manually define indexes using a `CREATE INDEX` statement. The same applies for the snowflake schema export.

Snowflake schema

The extension package provides the functionality to export an m-cube's facts as well as its non-dimensional data into a ROLAP data organization according to the snowflake schema. The ROLAP data organization according to the snowflake schema organizes the dimension tables within a (fully) normalized table hierarchy [Lan09]. Contrary to the simple star schema as presented previously, multiple tables per dimension are defined in order to represent non-dimensional data in the ROLAP snowflake organization. These

tables are connected hierarchically through foreign key constraints. The number of data can be reduced through a decrease in redundancy. As opposed to the star schema organization, where each tuple contains non-dimensional data of all levels above its own, data organized within a snowflake schema has no redundant information. For example, within the dimension table of the *location* dimension according to the star schema (table 4.20) names of superior aggregation levels as well as non-dimensional information, e.g., the *inhabitants* of a *city*, are stored redundantly within the table. The redundancy of non-dimensional data is aggravated when the approach of introducing a level attribute to store measures at different aggregation levels is employed within the ROLAP star schema, since the number of tuples is massively increased.

Each dimension table in the snowflake schema corresponds to a particular level within a dimension. Each tuple within a dimension table corresponds to an m-object with a top-level that equals the dimension table's level. Information of m-objects with the same top-level are grouped within the same dimension table. The dimension table hierarchy reflects the m-object hierarchy of a dimension. The naming scheme of the dimension tables is as follows: ⟨*name of the dimension*⟩⟨*user-defined suffix string*⟩_⟨*level*⟩. For example, table `location_dim$snow_store` holds non-dimensional data of m-objects contained in the *location* dimension with top-level store.

Heterogeneities in the level-hierarchy are handled by introducing dummy levels represented by dummy tuples. For example, m-object *Car* in the *product* dimension introduces an additional level *brand*. Only descendants of m-object *Car* have this level defined within their level-hierarchy. M-objects under *Book* do not define the *brand* level. M-objects *DaVinciCode* (descendant of *Book*) and *FiatPunto55* (descendant of *Car*) both have top-level *model*. These m-objects are stored within the same snowflake dimension table: table `product_dim$snow_model` in the example depicted in figure 4.5. The entries of this dimension table reference tuples of table `product_dim$snow_brand`. This table contains information of m-objects with top-level *brand* which is the parent-level of *model* within the *product* dimension's global level-hierarchy. However, descendants of m-object *Book* do not have an ancestor m-object with top-level *brand*. In this case, a dummy tuple that corresponds to the next ancestor m-object is created (see figure 4.6). For instance, a dummy *brand* named *Book* is inserted into the *brand* level dimension table and referenced by the entry that corresponds to m-object *DaVinciCode* in the *model* level dimension table. The level-specific attributes are `NULL` for the dummy tuple.

The fact tables in the snowflake schema employed by the extension package are comparable to the fact tables created for the star organization. In order to store heterogeneous measures within a snowflake schema, multiple fact tables have to be created – one fact table for each connection-level. Since dimension tables within a snowflake schema only contain m-objects at the same top-level, i.e., non-dimensional information of the same aggregation level, multiple fact tables have to be created in order to unambiguously reference tuples in the dimension tables through foreign key constraints. The naming scheme for fact tables is as follows: ⟨*name of the m-cube*⟩⟨*user-defined suffix string*⟩_⟨*shortened string representation of the connection-level*⟩; if the names are ambiguous, a running number is appended. The shortened string representation follows the same scheme as known from the measure tables. The shortened string representation of connection-levels

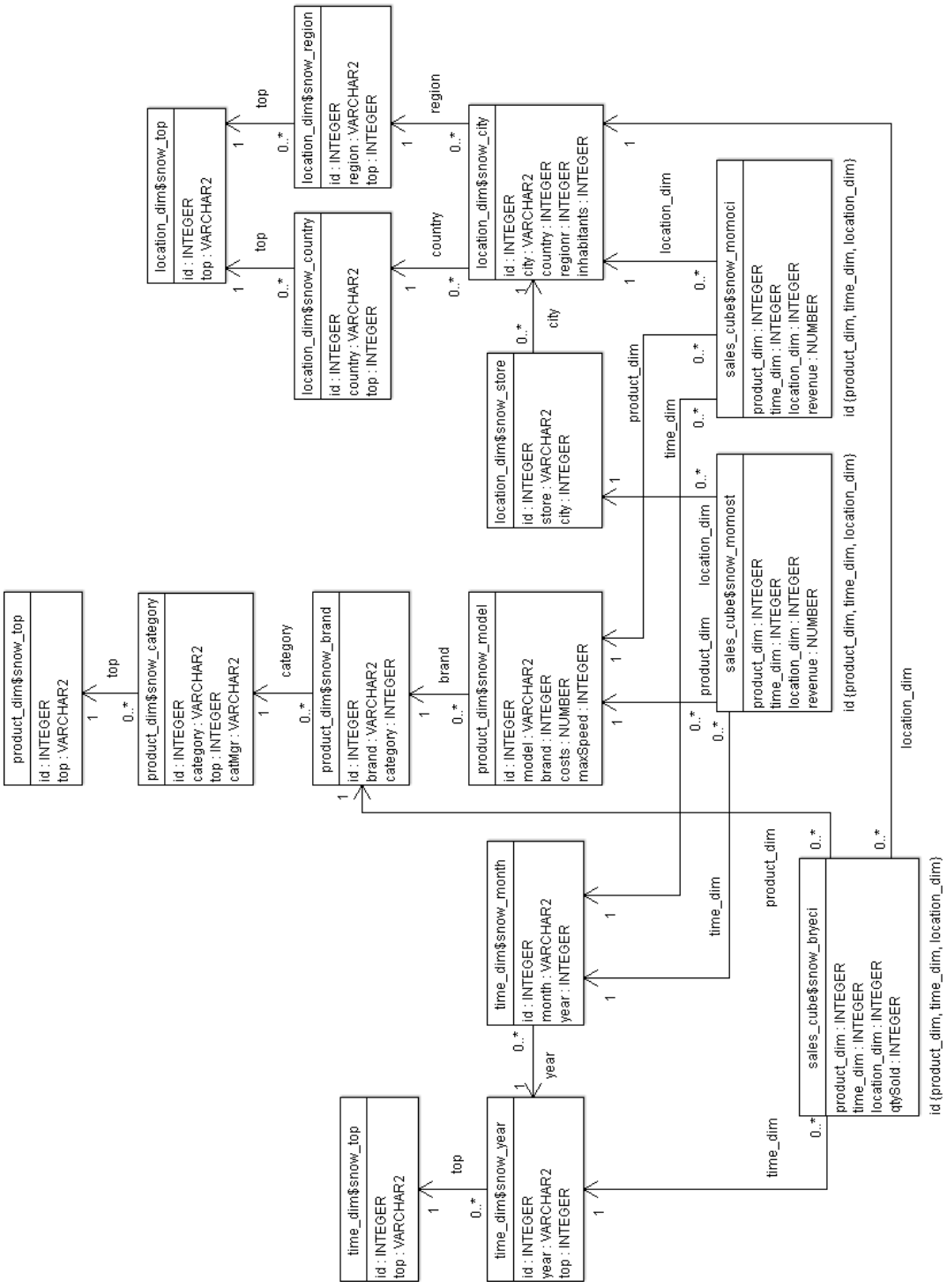


Figure 4.5: Example ROLAP organization according to the snowflake schema

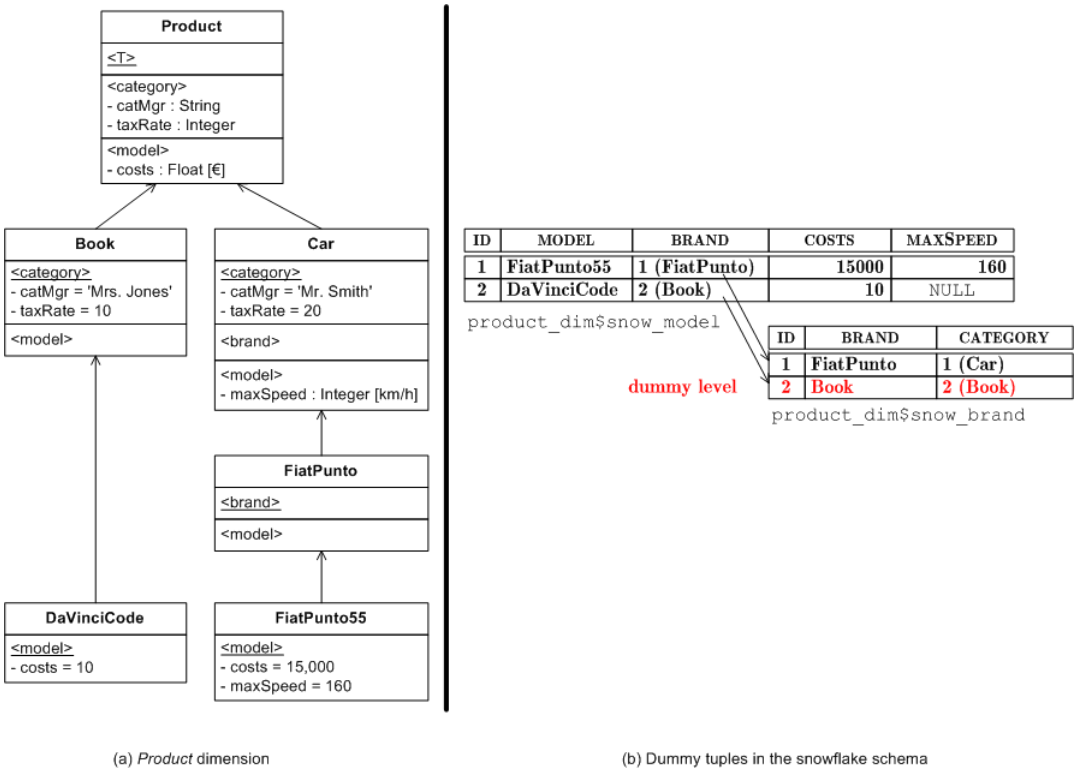


Figure 4.6: Example of a dummy tuple at level *brand* in the *product* dimension

is formed of the first two characters of the level names that define the connection-level. The employed organization of fact tables actually corresponds to the *fact constellation* schema (cf. [EN07], p. 984).

The complexity of the snowflake table organization is increased over the star schema. The number of tables as well as the number of foreign key references between these tables grow disproportionately with the number of dimensions. The complexity is further raised by introducing heterogeneities in the aggregation paths and the aggregation levels of measures. Table 4.5 illustrates how a three-dimensional, hetero-homogeneous m-cube might be exported into a ROLAP data organization according to the snowflake schema. Notice that multiple fact tables are created. These fact tables reference different dimension tables depending on the granularity of the contained measures. For example, fact table `sales_cube$snow_bryeci` stores measures at connection-level $\langle brand, year, city \rangle$ and consequently references the dimension tables that correspond to the respective level of granularity. Foreign key constraints ensure referential integrity and, in case of heterogeneous facts, can only be employed when multiple fact tables are created: In the snowflake schema it is not possible to store measures at different levels of granularity within a single fact table and have the referential integrity ensured by foreign key constraints.

4.3 Implementation details

In this section, the internal functioning of the implementation is explained. The functionality of the object types and their member methods is presented from the developer's point of view. The outline of this chapter loosely follows the outline of chapter 3. Notes on the methods' internal functioning are presented by task, i.e., the necessary steps to perform a certain task are explained from an internal perspective. This section illustrates how the m-cube's structure is defined and OLAP operations are performed – with the focus on how user calls to the interface are processed internally.

Auxiliary packages form an integral part of the extension package's internal architecture. A large part of the functionality, notably consistency checks and level-hierarchy operations, as well as dynamic object type creation, is delegated from the object type member methods to auxiliary PL/SQL packages. A description of the implementation's auxiliary packages is given in section 4.4.

4.3.1 Creating dimensions

Dimensions are created using the `create_dimension` function of package `mcube`. The procedure takes the unique name of the dimension and returns an object of type `dimension_ty`. Internally, the `create_dimension` function delegates most of the initialization work to the constructor function of `dimension_ty`. After the creation of the new dimension object it is inserted into the `dimensions` table. If the `dimensions` table does not exist at the time of the dimension's creation, it is created using dynamic SQL.

Dimensions are represented as objects of type `dimension_ty`. Unlike the object types representing the concepts of m-cubes, m-relationships, and m-objects, type `dimension_ty` is not dynamically concretized. This facilitates the task of function

`create_dimension` of package `mcube` since no object type has to be created using dynamic SQL. Most of the initialization work is done in the constructor function of object type `dimension_ty`. Function `create_dimension` takes over the part of updating the `dimensions` table, i.e., inserting the newly created dimension object into this table.

The constructor function of object type `dimension_ty` dynamically creates the concretization of `mobject_ty` that is used to create the dimension's m-object table. The name of this table is equal to the dimension's unique name. The name of the created m-object table is stored explicitly in the `mobj_table` attribute of `dimension_ty`. Internally, the creation of a dimension is a rather simple process compared to the handling of m-objects and the maintenance of m-cubes.

4.3.2 Handling m-objects

Attributes are handled by the member functions and procedures of `mobject_ty` and `mobject_*_ty`. Some methods are incorporated by the object type `dimension_ty`, e.g., adding m-objects to dimensions, retrieving m-objects from dimensions. The most complex tasks are related to the maintenance for attributes. It involves a time-consuming iteration through the dimensions m-object hierarchy.

Three topics can be identified with respect to the maintenance of m-objects. First, m-objects have to be created and added to the dimensions. The according functionality is incorporated in object type `dimension_ty` as well as the functions and procedures of `mobject_ty` and `mobject_*_ty`. Second, the maintenance of attributes, which is certainly the most complex task related to m-objects. Finally, attribute metadata has to be handled, which involves some error checking across the m-object hierarchy.

Adding m-objects to dimensions

M-objects are added to dimensions using function `create_mobject` in object type `dimension_ty`. The function creates an m-object of the concretized type `mobject_*_ty`, i.e., the m-object type representation for the particular dimension. This is done using dynamic SQL since `dimension_ty` is not concretized and does not know of the subtype to `mobject_ty` at compile-time. The newly created m-object is automatically inserted into the dimension's m-object table.

Type `dimension_ty` furthermore provides functions for retrieving a specific m-object or a reference to it; it also provides functions for retrieving collections of m-objects and m-object references. All of these functions use dynamic SQL since the name of the m-object table is not known at compile-time. For future releases, the possibility of concretizing type `dimension_ty`, in order to improve performance and delegate consistency checks to the database system, might be worth considering. The dynamic concretization of `dimension_ty` would enable the m-object table name to be hard-coded into the dimension body and thus spare some dynamic SQL calls which are more resource-intensive than native SQL.

When an m-object is created, the new m-object's list of ancestors has to be calculated. This functionality is encapsulated within function `calculate_ancestors` in object type

`mobject_ty`. Nested table `ancestors` containing an m-object's ancestors at their respective top-level has been introduced for the sake of efficiency. It is filled at the time of the m-object's creation. Function `calculate_ancestors` basically takes the `ancestors` nested table of the m-object's parents, copies the entries to the m-object's own `ancestors` nested table – disregarding duplicates – and adds entries for the m-object's parents.

Handling attributes of m-objects

Attributes of m-objects need to be defined before their value can be set. This is done using the `add_attribute` procedure defined by object type `mobject_ty`. Before a value is added, consistency checks are conducted. This task is delegated in parts to the auxiliary package `consistent_dimension`. If the attribute is already inherited or defined, an error is thrown. The `add_attribute` procedure's main responsibility, however, is to create an attribute table – if needed – where the attribute values can be stored.

Procedure `add_attribute` first determines whether there already exists an attribute table for attributes at this level, introduced by the particular m-object. This is done by iterating through the m-object's nested table `attribute_tables` which contains the names of the attribute tables defined by an m-object. For each attribute table the associated level is stored. If this level matches the level of the attribute that is to be introduced, the procedure does not need to create a new attribute table. Otherwise, a new attribute table is dynamically created by procedure `add_attribute`.

The length of table names in the Oracle database is restricted to 30 bytes. Thus, a table name cannot contain more than 30 characters. The naming scheme for attribute tables now is as follows: ⟨*name of dimension*⟩_⟨*name of introducing m-object*⟩_⟨*level*⟩. The naming scheme is intuitive and understandable for humans. However, the table name is likely to exceed a length of 30 bytes. The name of the dimension has to be included into this scheme since m-object names are only unique within one dimension. The name of an attribute table is consequently chopped off if it exceeds a length of 30 bytes. The string is chopped off at character position 25 and a number is appended. This number is used to distinguish between conflicting table names. Whether a table name conflicts with an existing table is detected by the `add_attribute` procedure by reading table `user_tab_columns`. Table `user_tab_columns` is a system table in the Oracle database and contains all tables defined by the user. The appended number is increased until a table name is found that is not in use already.

When an attribute's asserted value or metadata is set using the `set_attribute` procedure, the procedure has to determine whether the attribute is introduced by the m-object or inherited from an ancestor m-object. If an asserted value is to be set, the procedure furthermore needs the name of the table where values for this attribute are stored. When metadata is set, the attribute's level is needed as additional information. Both tasks need to obtain information whether the m-object even defines the attribute whose value is to be set – be it metadata or an asserted value.

Function `has_attribute` provides the functionality to determine if an attribute is inherited or defined by an m-object; it also obtains some additional information about an attribute. The query can be further specified such that only attributes at the m-object's

top-level or only attributes introduced by the m-object itself are considered, which is important when setting asserted values. The function returns `TRUE` if the m-object has the attribute in question, provided all restriction criteria are met. The query restriction on attributes at the m-object's top-level is necessary in order to find out whether an asserted value can be set, since m-objects can only assert a value for attributes at its top-level. Function `has_attribute` returns additional information about the attribute in question as output parameter.

The `has_attribute` function iterates through the m-object's nested table that stores the names of the attribute tables (`attribute_tables`). If the respective attribute table has a column bearing the name of the attribute in question, the m-object contains this particular attribute and the function returns `TRUE`. If only attributes at the m-object's top-level are searched (parameter `toplvl_only = TRUE`) the function only considers attribute tables for the m-object's top-level, i.e., attribute `lvl` of an entry in nested table `attribute_tables` is equal to the m-object's top-level. If the attribute is not found in one of the attribute tables defined by the m-object itself – i.e., in the m-object's `attribute_tables` nested table – the function continues its search in the attribute tables defined by ancestor m-objects. The ancestor m-objects' attribute tables are not included in the search if no inherited attributes should be considered (parameter `introduced_only = TRUE`).

By issuing an SQL `SELECT` statement on the `user_tab_columns` table, function `has_attribute` determines whether the attribute in question exists within a particular attribute table. The built-in table also contains metadata about tables, such as columns and their data types. This is the only way information about an m-object's attributes can be retrieved. Since column names are not case sensitive, attribute names have to be unique when ignoring the case. There is no other place where attributes are stored. The attribute's level is obtained from the nested table. Apart from finding out if a particular attribute table contains a given m-object attribute, table `user_tab_columns` is also used to construct an m-object attribute's description, represented by object type `attribute_ty`. Information about an attribute's data type is thus obtained. Since all attributes from the same table share the same level, the function can obtain the attribute's level from nested table `attribute_tables` in `mobject_ty`. The `list_attributes` function closely resembles the `has_attribute` function; its proceeding follows the approach of `has_attribute`.

Attribute values are set using procedure `set_measure` implemented by `mobject_ty`. The procedure checks whether an attribute with the given name is defined or inherited by the respective m-object; it further needs the name of the attribute table where the value has to be stored. This task is taken over by function `has_attribute`. Function `init_attr_table` is called in order to initialize the attribute table for the particular m-object, i.e., a tuple for the m-object is inserted where the m-object's attribute values are going to be stored. Procedure `set_measure` may then update this tuple using dynamic SQL. Dynamic SQL has to be used since the name of the attribute table is determined at run-time. Once set, a value can be retrieved from the attribute table by calling function `get_attribute`. The attribute's value is returned as an object of type `ANYDATA`. The

attribute table's name for the desired attribute is obtained from function `has_measure`. The value is retrieved from the attribute table by performing a dynamic SQL call.

A drawback of data type `ANYDATA` provided by the Oracle database in order to store values of an arbitrary type appears when inserting the attribute values into strongly typed relational tables. The attribute's value passed to procedure `set_measure` is of type `ANYDATA`; it has to be converted before being inserted into the attribute table. The value of `ANYDATA` converted to the original type can be accessed using functions `get*` and `access*`, e.g., `getVarchar2` or `accessVarchar2`. Both functions can be used to retrieve the value with the difference that `accessVarchar2` can be used in SQL queries (see [Ora09], [Moo09] for further information on `ANYDATA`). The implementation of the extension package consequently has to provide an alternative execution path for each possible data type. The same issue occurs when retrieving an attribute's value using function `get_attribute`. In this case, the attribute's value has to be converted into an object of type `ANYDATA` using function `convert*`, e.g., `convertVarchar2` or `convertNumber`.

Handling metadata about attributes

The overloaded procedure `set_attribute` is used to set attribute metadata. The alternative signature of procedure `set_attribute` demands the user to specify the attribute's name, a meta level, whether the value is shared or default, and the attribute's (meta-)value. Since the overloaded procedure can also be used to assert an attribute's value, the procedure delegates the task of setting the value to the alternative procedure `set_attribute` in case that the meta level is `NULL` and the passed value has not been marked as default.

Before a metadata entry can be inserted into the `attribute_metadata` nested table, procedure `set_attribute` has to verify that the m-object actually inherits or defines the attribute whose meta-value is to be set. This task is again delegated to function `has_measure`. Furthermore, the `set_attribute` procedure iterates through the ancestor m-objects' `attribute_metadata` nested table in order check whether there exists an entry at the same meta level marked as shared; in this case, an error is thrown. Afterwards, the procedure searches the metadata nested table for an existing tuple, if any, where to store the metadata. If no tuple exists, the nested table is extended by appending a new object of type `attr_meta_ty`. The attribute's level is stored within the `attribute_metadata` nested table as well, which is redundant information. This redundancy, however, may give way to indexing the metadata entries by level – an advantage not exploited in the current version of the prototype.

4.3.3 Creating m-cubes

M-cubes are created using the `mcube` package. The dynamic creation of object type `mcube_*_ty` and its dependent types (`mrel_*_ty`, `mrel_*_ty` etc.) is delegated to the `create_types` package. After the creation of the object types, the function assumes the task of constructing through dynamic SQL a coordinate object out of the passed m-object reference list in order to represent the new m-cube's root-coordinate. The constructor

function of object type `mcube_*_ty` contains no functionality but assigning the parameter values to the attributes.

The creation of m-cubes has to be generic. The `mcube` package is the starting point for the concretization of the object types used to represent the hetero-homogeneous data warehouse. It has to provide a generic interface that can be used to construct m-cubes. Function `create_mcube` in the `mcube` package accepts a list of dimension object references or, alternatively, a list of dimension names and concretizes the m-cube object type as well as the related types

4.3.4 Handling m-relationships

M-relationships are handled by the member functions and procedures of `mcube_*_ty` and `mrel_*_ty`. M-relationships are closely tied to the auxiliary object types `coord_*_ty` and `conlevel_*_ty`. Some functionality is thus incorporated by these object types, e.g., the partial order of coordinates implies the partial order of m-relationships.

Adding m-relationships to m-cubes

M-relationships are always cube-specific. They are created by calling an m-cube's `create_mrel` function which creates an m-relationship of the concretized and cube-specific object type `mrel_*_ty`. Function `create_mrel` is dynamically created and demands as parameters the references to the m-objects that constitute the m-relationship's coordinate.

A coordinate is the identifier for an m-relationship. The m-cube's m-relationship table, however, does not define a primary key constraint. This is due to fact that the Oracle database does not allow to define primary keys on objects or references (ORA-02329). Thus, the `create_mrel` function checks for duplicate entries in the m-relationship table and throws a custom error code in case of a violation.

The constructor function of `mrel_*_ty` is customized for each m-cube as well. Each m-relationship stores a reference to the m-cube it belongs to. The constructor function only takes a reference to an object of the concretized m-cube object type (`REF mcube_*_ty`). Each component of the m-relationship's coordinate is passed to the constructor function as a separate argument. The references to the respective coordinate m-objects point to objects of the concretized type `mobject_*_ty`.

The coordinate of an m-relationship defines its relative order to other m-relationships of the m-cube. Order function `compare_to` of object type `coord_*_ty` is used to find out an m-relationships relative order to another m-relationship. This function can also be used to retrieve an ordered set of m-objects out of the database in SQL; an ordered set of m-relationships is needed by the measure-handling methods.

Retrieving an m-relationship's ancestors is a costly task to perform. The ancestors of an m-relationship are needed by most OLAP operations as well as for consistency checking and measure retrieval. Computing the partial order of m-relationships is time-consuming and considerably slows down the execution of OLAP queries. Future implementations will have to improve this situation by storing the ancestors of m-relationships in a similar

way as an m-object's ancestors are stored redundantly. This will possibly improve the speed of OLAP operations. Large potentials in performance improvements have not yet been exploited by the prototype.

Handling measures in m-relationships

Measures are added to m-relationships using the `add_measure` procedure of m-relationships. Measures are stored analogously to m-object attributes. Adding measures to m-relationships closely resembles the act of adding attributes to m-objects. The `add_measure` procedure assumes the task of creating or, alternatively, alter the measure table and adding a tuple to the m-relationship's nested table containing the name of the newly created measure table.

Function `has_measure` of `mrel_*_ty` works similar to the `has_attribute` function of `mobject_ty`. It is used to check whether a particular m-relationship introduces or inherits a particular measure. This is done by iterating through nested table `measure_tables` and querying system table `user_tab_columns`. When only measures introduced by the m-relationship are to be considered, function `has_measure` only considers measures for which the respective m-relationship has an entry in its `measure_tables` nested table. This can be done since an m-relationship only creates a new measure table for a measure it originally introduced. An exception to this rule occurs when an m-relationship moves a measure introduced by one of its ancestors to a more specific level of aggregation. In this case, however, the extension package counts such a measure as introduced by the m-relationship that moves the measure to the more specific level of aggregation.

The `list_measures` function of `mrel_*_ty` works similar to the `has_measure` function. It returns a list of objects of type `measure_*_ty` containing information about a particular measure. These information include the name of table where the measure's value is stored, the measure's connection-level as an object of type `conlevel_*_ty` and information about the measure's data type. The `list_measures` function is massively used by the methods implementing the OLAP query operations for m-cubes.

Handling metadata about measures

Metadata about measures is handled similar to m-object attribute metadata. The overloaded procedure `set_measure` is used to set attribute metadata. The alternative signature of procedure `set_measure` demands the user to specify the measure's name, a meta level, whether the value is shared or default, and the measure's (meta-)value. Since the overloaded procedure can also be used to assert an measure's value, the procedure delegates the task of setting the value to the alternative procedure `set_measure` in case that the meta level is `NULL` and the passed value has not been marked as default.

In addition to a measure's unit, the aggregation function is stored as metadata. The aggregation function of a measure is set like any other measure metadata. However, in order for query functions to automatically recognize a measure's aggregation function, the storage has to follow certain semantics. The semantics of how to store a measure's aggregation function has not yet been fully determined. How-

ever, function `set_aggregation_function` encapsulates the functionality to store a measure's aggregation function within the `measure_metadata` nested table. Function `get_aggregation_function` retrieves a measure's aggregation function.

Function `get_aggregation_function` returns the aggregation function for a given measure. The aggregation function for a measure can only be defined by the m-relationship that originally introduced the measure. In order to guarantee the stability of aggregation functions, other measure's may not change the aggregation function, even if they move the measure to a more specific level of aggregation. Function `get_aggregation_function` retrieves the m-relationship that originally introduced the given measure by issuing an SQL statement. Function `has_measure` considers measures where a particular m-relationship moves the measure to a more specific connection-level as introduced by this m-relationship. The query in `get_aggregation_function` has to be restricted such that only m-relationships are considered where there does not exist another m-relationship that introduces the same measure at a more general level of granularity. Function `get_aggregation_function` expects the aggregation function to be stored at meta level *function*. The measure's meta-value – stored as `ANYDATA` – is expected to be a string representation of the aggregation function; future releases might use m-objects to represent aggregation functions.

4.3.5 OLAP with hetero-homogeneous hierarchies and cubes

OLAP operations are the most time-consuming activities performed by the prototypical extension package. Depending on the employed query semantics, iteration steps and consistency checks vary in number and thus the semantics considerably influence operation time. While object-preserving queries as employed in query views are generally less-expensive in terms of run-time and memory consumption, object-generating queries demand more effort for the computation of the result. Future implementations will have to be optimized with respect to the query operations provided by the extension package.

Creating flat data warehouse schemata

Star and snowflake schema export functionality is divided into the export of dimensions and the export of the m-cube's m-relationships and their measures. Type `dimension_ty` provides procedures `create_star` and `create_snowflake` which incorporate the functionality to create dimension tables according to the star and the snowflake schema, respectively. Procedures `create_star` and `create_snowflake` of type `mcube_*_ty` encapsulate the functionality to export an m-cube's m-relationships and measures into fact tables. The export procedures of `mcube_*_ty` depend on those provided by object type `dimension_ty`.

Procedures `create_star` and `create_snowflake` of `dimension_ty` each have a dynamic anonymous PL/SQL block which in turn dynamically constructs the create and insert statements to define and fill the dimension table(s). The dynamically executed PL/SQL block in both procedures loop through the dimension's set of m-objects in order to create the dimension table(s). Within the dynamically executed PL/SQL block,

another string which contains insert statements is created dynamically. Two levels of dynamic SQL are featured.

Procedure `create_snowflake` has to introduce *dummy levels* in case of alternative aggregation paths. For example, products in general do not have aggregation level *brand*; there is no *brand* for books. However, cars may well have an additional level *brand*. In this case, a dummy level is inserted for all books. All books belong to the same dummy *brand*, namely *Book*. In order to achieve this, the dynamically executed PL/SQL block in procedure `create_snowflake` makes use of an inner function `insert_data` that introduces dummy levels recursively. The inner function adds dummy levels for each level not defined by an ancestor of a particular m-object as long as this level is contained within the dimension's global level-hierarchy.

Procedures `create_star` and `create_snowflake` of type `mcube_*_ty` call the respective procedures of type `dimension_ty` in order to create the dimension tables. The fact tables are the created by iterating through the m-cube's set of m-relationships. Insert statements are dynamically created and executed. Procedure `create_snowflake` has the more complex task as multiple fact tables have to be created – one for each level of aggregation.

Procedure `create_snowflake` of type `mcube_*_ty` locally defines the record type `table_level_mapping_ty` storing the name of a fact table together with its level of aggregation represented by an object of type `conlevel_*_ty`. Procedure `create_snowflake` iterates through the m-cube's set of m-relationships and retrieves for each m-relationship a list of measures at the m-relationship's top-level, i.e., the measure for which a value is asserted by the m-relationship. The procedure loops through the list of measures, retrieves each measure's value, and inserts a tuple into the corresponding fact table. The fact table for a particular name is obtained from a list of `table_level_mapping_ty` objects. If no fact table exists for a particular level of aggregation, a new fact table is created.

Fact extraction and aggregation of measures

Function `value_aggregation` in `mcube_*_ty` can be used to perform an aggregation of a particular measure of an existing m-cube. The function applies for the measure the aggregation function that has been defined in the `measure_metadata` nested table of the m-cube's m-relationship. Auxiliary function `get_aggregation_function` has been defined for object type `mrel_*_ty` in order to retrieve the aggregation function for a particular measure.

In order to perform a roll-up for a given measure at a particular coordinate within the m-cube, function `value_aggregation` retrieves all m-relationships at descendant coordinates of the aggregation coordinate (or at the aggregation coordinate). Only m-relationships are considered that inherit or define the desired measure at their top connection-level. If function `get_aggregation_function` retrieves no aggregation function (`NULL`), it is assumed to be `SUM`. Function `value_aggregation` loops through the set of the m-relationships defined at coordinates which are descendants of the roll-up co-

ordinate and apply the given aggregation function on the values. The result is converted to type `ANYDATA`.

Function `fact_extraction` performs an aggregation of values for a given roll-up coordinate on multiple measures. The result is written to a table that contains only one tuple. The fact extraction table is created dynamically. The function relies on the `value_aggregation` function; it is called for each measure at the same roll-up coordinate. The `ANYDATA` value is then converted into a primitive data type and inserted into the result table.

Branching dimensions

The branch functionality is a preliminary for dimension object-generating queries. Function `branch` of object type `dimension_ty` extracts a sub-branch of the dimension's m-object hierarchy and creates a new dimension with independent copies of the original m-objects. The function makes use of the dynamic SQL in order to retrieve all m-objects of the sub-branch from the dimension's m-object table as well as for retrieving references from the new dimension. The SQL query as shown in listing 4.1 is used to retrieve all m-objects below or equal to the root m-object of the *product* dimension's branch. The example illustrates how an *Alps* branch might be extracted from the *product* dimension using this query.

Listing 4.1: SQL query to branch the *product* dimension

```
1  DECLARE
2    /* the name of the new dimension's root m-object */
3    root_oname := 'Alps';
4  BEGIN
5    /* get the 'Alps' branch of the product dimension */
6    SELECT VALUE(o)
7    FROM   product_dim o, TABLE(o.ancestors) p
8    WHERE  DEREF(p.ancestor).oname = root_oname OR
9           o.oname = root_oname
10   ORDER BY VALUE(o) ASC;
11 END;
```

After the execution of the branch SQL query, the retrieved m-objects have to be copied, with the level-hierarchy being adapted accordingly. The `branch` function creates independent copies of the original dimension's m-objects. Attributes defined by ancestors of the branch's root m-object are then introduced by the new dimension's root m-object. Levels that would violate the unique induction rule for levels are disregarded in the result dimension. The references to the parent m-objects need to be updated.

The SQL query shown in listing 4.1 orders the selected m-objects using the `compare_to` order function of `mobject_ty`. This function orders m-objects based on their top-levels and the precedence in the dimension's global level-hierarchy. The m-objects need to

be ordered since the implementation uses the m-object's object type functions used to add attributes and assign values to them. An attribute's value can only be set for an attribute that has been defined. When looping through the set of selected m-objects, in order to assign values for the m-objects' attributes, the set of m-objects has to be ordered. Furthermore, since the references to the parents of the new dimension's m-objects need to be updated, the m-objects have to be inserted in the right order.

The level-hierarchy of each m-object has to be updated. Due to the unique induction rule for levels and the fact that m-objects that are not descendants of the new dimension's root m-object, some aggregation paths are disregarded in the result dimension. Levels that have been introduced by an m-object that is not included in the branch – as well as their aggregation path – are not included and thus cleared from the m-object copy of the result dimension. If a particular m-object has a top-level that has been introduced by an m-object that is not included in the branch dimension, the m-object is disregarded in the result dimension.

Object-generating closed m-cube query operations

The closed query operations provided by object type `mcube_*_ty` are object-generating in the sense that new m-cube and m-relationship objects are created. The dimension object-generating mode is not fully supported by the current version of the prototype. The closed m-cube query operations that are dimension object-preserving, however, are operable. Nevertheless, these query functions are basically object-generating in their nature.

The object-generating query function `dice` iterates through the entries of the m-cube's m-relationship table. If the query function is dimension object-generating, the branch function of the m-cube's dimensions is called, thus creating independent branches of the original dimensions. This is only done if the dice coordinate m-object is a concretization of the root m-object in the respective dimension. If the `dice` function is executed in dimension object-preserving mode, the m-objects in the original dimension are referenced.

Function `dice` has to move some of the m-relationships of the original m-cube to a different coordinate. For each m-relationship, the coordinate is moved to a different coordinate if one or more of the m-objects that make up the original m-relationship's coordinate is an ancestor of the respective m-object in the new root-coordinate. For example, an m-relationship in the original m-cube is defined at coordinate (*Product*, *Year2010*, *Alps*) and introduces measure *cheapestOffer*. The user applies a *dice* on this *sales* cube in order to retrieve car sales in the year 2010 in the *Alps* region, i.e., the dice coordinate is (*Car*, *Year2010*, *Alps*). Consequently, the result m-cube has the measure *cheapestOffer* defined by the m-relationship at the coordinate (*Car*, *Year2010*, *Alps*). At each iteration step, function `dice` checks for each dimension whether the coordinate m-object is a descendant or ancestor of, equal to or in no concretization relationship with the m-object of the new root-coordinate in the respective dimension. The outcome of this `IF` statement determines the coordinate of the m-relationship in the result m-cube.

Some of the m-cube *dice* functionality has to be executed dynamically. For inserting the m-relationships into the result m-cube, the procedures of the concretized `mcube_*_ty` ob-

ject type have to be accessed. This can only be done dynamically since the `create_mcube` function of package `mcube` is a generic implementation and consequently returns an instance of the abstract super type `mcube_ty`. Measures are also added to the m-relationships of the result m-cube using dynamic SQL. A measure is only included if each of the levels that compose the measure's connection-level is contained within the level-hierarchy of the m-object in the respective dimension of the coordinate.

The `slice` function first determines for each m-object of each dimension if the m-object satisfies the predicate of the respective dimension. Since function `satisfies` only returns `TRUE` for an m-object whose top-level equals to the level the predicate of the dimension has been defined upon, after determining the m-objects that satisfy the predicate the ancestor and descendant m-objects of the satisfying m-objects have to be retrieved. This is done using the query as presented in listing 4.2. Listing 4.2 illustrates how all m-objects of the *product* dimension that are in a concretization relationship with a given m-object are retrieved. Only m-relationships that connect only m-objects contained in one of these sets are included in the result m-cube.

Listing 4.2: SQL query for selecting all m-objects that are in a concretization relationship with a given other m-object

```
1  SELECT d.obj_ref
2  FROM   (SELECT REF(d1) AS obj_ref, VALUE(d1) AS val
3          FROM   product_dim d1,
4                 product_dim d2,
5                 TABLE(d2.ancestors) d3
6          WHERE  d2.oname = name of m-object AND
7                 REF(d1) IN d3.ancestor
8
9          UNION
10
11         SELECT REF(d1) AS obj_ref, VALUE(d1) AS val
12         FROM   product_dim d1
13         WHERE  d1.oname = name of m-object name
14
15         UNION
16
17         SELECT REF(d1) AS obj_ref, VALUE(d1) AS val
18         FROM   product_dim d1,
19                product_dim d2,
20                TABLE(d1.ancestors) d3
21         WHERE  d2.oname = name of m-object name AND
22                REF(d2) IN d3.ancestor) d
```

The object-generating `project` function is straight-forward and arguably the least complex closed m-cube query operation. No dimension object-generating option exists

for the `project` function. M-relationships are copied to the result m-cube as is, with the exception of the included measures. Only the measures passed to the function are included in the m-relationships of the result m-cube.

Object-preserving query views

Performing query operations using the object-preserving query views is less time- and space-consuming than their object-generating counterparts. Query views only maintain a set of references to m-relationships as well as a list of measures. No costly operations are applied for copying m-relationships and measures in order to obtain independent copies of the originals or alter m-relationships so as to obtain a consistent result m-cube. This makes using query views the preferred way of querying m-cubes.

A list of query expressions is maintained by each query view. These expressions describe a particular query operation. For each expression type – i.e., dice, slice, and project – a separate evaluation function `evaluate` exists which encapsulates the actual query functionality. Each of these functions alters the query view's set of m-relationships and/or measures. The no-parameter version of function `evaluate` evaluates the expression list by calling the overridden versions of `evaluate`. The query view's `dice`, `slice`, and `project` functions do not immediately execute the query operation. Rather, the corresponding query expression is added to the list of expressions.

The evaluation of the query expressions is similar to the execution of the object-generating closed query operations. However, most of the code that is needed to alter m-relationships and move measures between them in order to obtain a consistent result dimension can be omitted. The export functions `create_star` and `create_snowflake` as well as the fact extraction and value aggregation functions have to be altered so that only the query view's set of m-relationships and measures are considered. The algorithms, however, have to be only moderately modified.

4.4 Auxiliary packages and types

Some functionality is encapsulated in auxiliary packages. Functionality used by multiple methods defined by the various object types is delegated to auxiliary PL/SQL packages in order to master complexity and increase the maintainability of the code. Consistency checking and operations for analyzing and manipulating level-hierarchies are used both for defining and querying m-cubes. Other packages encapsulate the functionality to dynamically create object types and triggers. This section briefly introduces the auxiliary packages of the extension package.

4.4.1 Package `create_types`

Package `create_types` encapsulates most of the functionality to dynamically create the concretized object types. With the exception of `mobject_*_ty` which is created in the constructor of type `dimension_ty`, all object type concretizations are dynamically created by this package. All created types are m-cube dependent. They are concretized

with respect to the number of dimensions along with the creation of the m-cube. Each procedure of the `create_types` package has a list with the m-cube's dimension names passed. The object type source code is then constructed dynamically.

4.4.2 Package `create_triggers`

Package `create_triggers` encapsulates the functionality to dynamically create triggers. In the current version, only one trigger is created which is used to update the dimension's global level-hierarchy each time a new m-object is inserted. In future versions, some consistency checks might be realized using triggers, e.g., to prevent the dimensions of an m-cube from being altered.

4.4.3 Package `levelhierarchies`

Package `levelhierarchies` is the most widely used auxiliary package. It incorporates all functionality which is needed to analyze an m-object's level-hierarchy. The package is used in virtually all aspects of the implementation. It furthermore provides functionality to manipulate an m-object's level-hierarchy, e.g., deleting a particular aggregation path from an existing level-hierarchy.

The package's functions and procedures are used whenever the precedence of levels is checked or whether a level-hierarchy contains a particular level. The package can also be used to determine a particular level's parent-levels or if a level has sub-levels. Function `normalize` is used to eliminate duplicate entries – this is important for updating the dimension's global levelhierarchy.

Function `clear_path` is used to delete an aggregation path. This is used for obtaining a branch of a dimension. All entries involving the given level either in the `lvl` or `parentlevel` column of the level-hierarchy are deleted as well. Furthermore, the `clear_path` function removes all sub-levels of a given level from the level-hierarchy if they have no other parent-levels.

4.4.4 Package `collections`

Package `collections` is a minor auxiliary package used to facilitate the querying of elements in the `names_tty` table type. It is used to encapsulate a task which would otherwise have to be redundantly implemented. In future versions, this package might comprise additional functions and procedures to maintain and analyze various types of collections. Despite being collections in some sort of way, level-hierarchies are handled in a different package `levelhierarchies`, dedicated only to handle level-hierarchies (`p_tty`).

4.4.5 Packages for consistency checking

Consistency checks are largely delegated to three dedicated PL/SQL packages. Each package encapsulates consistency checks of similar consistency rules. Three types of consistency rules have to be distinguished. First, the consistent concretization of m-objects whose consistency check functionality is encapsulated within package

`consistent_mobj_concretization`. Second, the rules for the consistent dimension are incorporated within package `consistent_dimension`. Last, package `consistent_mcube` checks for violations of the rules for the consistent m-cube. Notice that the functions in the consistency checking packages return `BOOLEAN` values and do not throw an error. For a detailed description of the m-cube consistency rules the reader may refer to chapter 2.

Package `consistent_mobj_concretization` encapsulates functions to determine whether an m-object violates the rules for the consistent concretization of m-objects. Function `check_toplevel_consistency` checks whether the top-level of a particular m-object is a second top-level in its parents' level-hierarchies. This is done by looping through the m-object's parents. Function `check_level_consistency` checks whether an m-object's level-hierarchy contains all levels from the parent m-objects' top-level downwards. Function `check_hierarchy_consistency` determines if the level-hierarchies are of two m-objects within a concretization relationship are compatible with each other. This is determined by iterating through the parent m-object's level-hierarchy and verifying for each level that the transitive order of the two levels has not changed in the concretizing m-object. The locality of level order is checked by function `check_locality_lvl_order`.

Package `consistent_dimension` encapsulates functions to check for the unique induction rules for levels and attributes. Function `check_unique_attr_induction` checks if the dimension obeys the unique induction rules for attributes; function `check_unique_level_induction` checks for the unique induction rule for levels. Both functions involve iterating through the dimension's set of m-objects. The verification for a given m-object if a given level violates the unique induction rule for levels is done by the `check_unique_level_induction` function. If the given level is contained in none of the given m-object's parents' level-hierarchies, the function has to iterate through the dimension's set of m-objects. The check is only passed if no other m-object contains the level that is to be introduced in its level-hierarchy. Notice that function `check_unique_level_induction` only returns reasonable results at the time of the m-object's creation. When adding descendant m-objects that have to include the newly introduced level have been added, the function would report the originally introducing m-object as violating the unique induction rule for levels. The consistency check functions are generally designed to be enforced at the time of the definition of the m-cube's dimensional structure.

Package `consistent_mcube` only encapsulates a function that checks an m-cube for violations of the unique induction rule for measures. This is done by using dynamic SQL to iterate through the m-cubes set of m-relationships and check whether there has already been the same measure introduced by a different m-relationship. Moving measures to a more specific level of granularity is supported though.

In general, consistency checks considerably slow down the execution of the operations. Future implementations will have to provide a mechanism to deactivate consistency checks for the sake of efficiency. The proof-of-concept prototype, however, omits this option and always enforces consistency checks.

4.5 Error handling

Error codes ranging from -20000 to -20999 are provided for customized error messages by the Oracle database. Built-in procedure `raise_application_error` is used to raise error codes with a custom error message. Error messages are thrown by the m-cube extension package when consistency problems in an m-cube, dimension, m-object or m-relationship are detected.

In order to represent consistency errors, object type `error_ty` is used. An m-cube consistency error is consists of three elements. First, a unique ORA-code is assigned to each consistency problem. Second, a unique name is associated with each consistency problem. It is used to refer to a certain error in a human readable way. Third, a custom error message is assigned to an error type. This message is displayed to the user when the system encounters a consistency violation.

Error types are stored in a separate object table. Table `errors` is an object table of type `error_ty`. The primary key of this table is the error code. The error codes range from -20000 to -20999. Attribute `error_name` is unique as well. The error message cannot be longer than 2048 bytes, which is the constraint posed by the database for custom error messages.

Procedure `raise_error` in object type `error_ty` throws an ORA-code by calling the built-in procedure `raise_application_error` with the error code and message assigned to the object's respective attributes. When a consistency problem is encountered, the corresponding object from the `errors` object table is selected, using the unique name as identifier, and the object's `raise_error` procedure is called. This causes an error with the corresponding ORA-code being thrown.

Error codes can be classified in seven categories. Error codes starting with ORA-200xx denote inconsistent m-objects, e.g., when trying to set a value for an attribute that is not available for the particular m-object. Error codes starting with ORA-201xx are violations of the rules for consistent concretization of m-objects. Error codes starting with ORA-202xx signify inconsistencies within a dimension. Error codes starting with ORA-203xx denote inconsistent m-relationships, e.g., when trying to set a value for an attribute that is not available for the particular m-relationships. Violations of the rules for consistent concretization of m-relationships are indicated by error codes starting with ORA-204xx. Consistent m-cube error codes start with ORA-205xx. Miscellaneous error codes start at ORA-206xx; no errors are currently associated with this class. Table 4.22 gives an overview of the error codes defined by the extension package along with a description.

ORA-Code	Description
-20000	The attribute the user tried to set a value for does not exist for this m-relationship.
-20100	The top-level of a concretized m-object o' is not a second-top-level of its parent m-object o.
-20101	Level containment: A level of m-object o is not contained in the level-hierarchy of descendant m-object o'.
-20102	Level order compatibility: The relative order of levels is different in m-object o and its descendant m-object o'.
-20103	Locality of level order: M-object o' introduced a level that contains a parent-level that is not in its level-hierarchy.
-20104	Tried to change an attribute (meta-)value that was not default.
-20200	Unique induction rule for attributes: Attribute is already defined or inherited by the m-object trying to add the new attribute.
-20201	Unique induction rule for attributes: Attribute has already been inducted by another m-object.
-20202	Unique induction rule for levels: Level has already been inducted by another m-object.
-20300	The measure the user tried to set a value for does not exist for this m-relationship.
-20400	Assured granularity: M-relationship moves an already defined measure to a less detailed level of granularity.
-20401	Tried to change a measure (meta-)value that was not default.
-20500	The m-cube lacks a single m-relationship that corresponds to the m-cube's root coordinate.
-20501	Tried to insert an m-relationship into a cell where another m-relationship for that coordinate already exists.
-20502	Unique induction rule for measures: Measure is already defined by the m-relationship trying to add the new measure.
-20503	Unique induction rule for measures: Measure has already been introduced by another m-relationship.
-20504	This error-code signifies violations of the unique value assertion rule.
-20505	Tried to insert an m-relationship at a coordinate that is not a proper sub-coordinate of the m-cube's root-coordinate.

Table 4.22: Error codes

A Installation

The extension package for hetero-homogeneous dimension hierarchies and cubes was implemented in the Oracle database 11g. Information on the installation and configuration of the Oracle database on various platforms can be found in [ora].

A database user/schema has to be created for the extension package. The user needs to be assigned certain privileges in order for the extension package to work correctly. The following privileges are required by the user:

- Privileges `CONNECT` and `CREATE SESSION` are required for establishing a connection to the database.
- `CREATE PROCEDURE`
- `CREATE TABLE`
- `CREATE TYPE`
- `CREATE TRIGGER`
- The created user needs the privilege to execute the procedures and functions of the provided packages. This can either be done by granting the user the `EXECUTE ANY PROCEDURE` privilege (not recommended) or by granting `EXECUTE` privileges on all of the created packages in the user's schema.

The extension package's source code is shipped in four files. These files have to be run in a determined order for the packages and object types to compile correctly. The type and package definitions contained in one source file are arranged such that one source file can be run as is, without the user taking care of the execution order of the definitions within the file. However, the separate source files need to be compiled in the following order:

1. File *error_code.sql* creates type `error_ty` which is used for raising errors and sets up the `errors` table.
2. File *mobject.sql* defines type `mobject_ty` and its associated object types. Furthermore, auxiliary packages for m-object consistency checks and level-hierarchy queries as well as the header definition of `dimension_ty` are included in this file.
3. File *dimension.sql* defines the body of type `dimension_ty`.

4. File *mcube.sql* creates packages `mcube` and `create_types` which contain the logic for dynamically creating m-cubes and the corresponding types. The abstract basetype of m-cubes – `mcube_ty` – is defined as well as auxiliary types such as the query view and its related types.

After compiling the sources, the packages should be available in the database. The extension package prototype can then be used as described in chapter 3, provided the rights have been set correctly. The extension package can be re-installed over an existing installation by following the same steps. The user has to take care that all created cubes, dimensions, and related object types are deleted, either by using the provided deletion procedures or by manually dropping the created tables and types.

Bibliography

[AK00] C. Atkinson and T. Kühne. Meta-level independent modelling. In *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, pages 12–16, 2000.

[AK01] C. Atkinson and T. Kühne. The essence of multilevel metamodeling. *4th International Conference on UML – The Unified Modeling Language. (Lecture Notes in Computer Science)*, 2185/2001:19–33, 2001.

[AK09] Lance Ashdown and Tom Kyte. *Oracle Database Concepts, 11g Release 2 (11.2)*. Oracle Corporation, 2009.

[Ber08] Stefan Berger. Lecture notes in data warehousing. Course 'Data Warehousing' held at the JKU Linz, 2008.

[Bro01] P.G. Brown. *Object-Relational Database Development: A Plumber's Guide*. Prentice Hall, 2001.

[BTLM01] Robert M. Bruckner, A Min Tjoa, Tok Wang Ling, and Oscar Mangisengi. A framework for a multidimensional olap model using topic maps. *International Conference on Web Information Systems Engineering*, 2, 2001.

[Bul96] D. Bulos. A New Dimension. *Database Programming & Design*, 6:33–37, 1996.

[CA09] Immanuel Chan and Lance Ashdown. *Oracle Database Performance Tuning Guide, 11g Release 2 (11.2)*. Oracle Corporation, 2009.

[Cod90] E. F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[EN07] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems*. Addison Wesley, fifth edition, 2007.

[GMR98] Matteo Golfarelli, Dario Maio, and Stefano Rizzi. The dimensional fact model: a conceptual model for data warehouses. *International Journal of Cooperative Information Systems*, 7(2–3):215–247, 1998.

[GPHS06] Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software & System Modeling*, 5(1):72–90, 2006.

[Hah06] Michael Hahne. Mehrdimensionale Datenmodellierung für analyseorientierte Informationssysteme. In Peter Chamoni and Peter Gluchowski, editors, *Analytische Informationssysteme – Business Intelligence-Technologien und -Anwendungen*, pages 177–206. Springer Verlag, third edition, 2006.

[IBM] IBM. *DB2 Infocenter*. `http://publib.boulder.ibm.com/infocenter/imzic`; visited on February 11, 2010.

[Inm02] William H. Inmon. *Building the data warehouse*. John Wiley & Sons, third edition, 2002.

[JLVV03] Matthias Jarke, Maurizio Lenzerini, Yannis Vassiliou, and Panos Vassiliadis. *Fundamentals of data warehouses*. Springer Verlag, second edition, 2003.

[KR02] Ralph Kimball and Margy Ross. *The data warehouse toolkit*. Wiley, second edition, 2002.

[Lan09] Paul Lane. *Oracle Database Data Warehousing Guide, 11g Release 2 (11.2)*. Oracle Corporation, 2009.

[LT01] Hans-Joachim Lenz and Bernhard Thalheim. Olap databases and aggregation functions. In *SSDBM*, pages 91–100. IEEE Computer Society, 2001.

[LT09] H.-J. Lenz and B. Thalheim. A formal framework of aggregation for the olap-oltp model. *Journal of Universal Computer Science*, 15(1):273–303, 2009. `http://www.jucs.org/jucs_15_1/a_formal_framework_of`.

[Moo09] Sheila Moore. *Oracle Database PL/SQL Language Reference, 11g Release 2 (11.2)*. Oracle Corporation, 2009.

[Neu10] Bernd Neumayr. *Multi-Level Modeling with M-Objects and M-Relationships*. PhD thesis, Johannes Kepler Universität Linz, 2010. Working Draft.

[NGS09] Bernd Neumayr, Katharina Grün, and Michael Schrefl. Multi-level domain modeling with m-objects and m-relationships. In *Sixth Asia-Pacific Conference on Conceptual Modelling (APCCM)*, 2009.

[NN09] Marko Niinimäki and Tapio Niemi. An etl process for olap using rdf/owl ontologies. *Journal on Data Semantics*, 13:97–119, 2009.

[NS08] Bernd Neumayr and Michael Schrefl. Comparison criteria for ontological multi-level modeling. Technical Report 08.03, Johannes Kepler Universität Linz, 2008.

[NST10] Bernd Neumayr, Michael Schrefl, and Bernhard Thalheim. Hetero-homogeneous hierarchies in data warehouses. In *Seventh Asia-Pacific Conference on Conceptual Modelling (APCCM)*, 2010.

[OF98] J.J. Odell and M. Fowler. *Advanced object-oriented analysis and design using UML*, chapter Power types, pages 23–32. Cambridge University Press, 1998.

[ora] Oracle Database Online Documentation 11g Release 2. Website. `http://www.oracle.com/pls/db112/homepage`; visited on February 14, 2010.

[Ora09] Oracle Corporation. *Oracle Database Object-Relational Developer's Guide, 11g Release 2 (11.2)*, 2009.

[PCTM02] John Poole, Dan Chang, Douglas Tolbert, and David Mellor. *Common Warehouse Metamodel – An introduction to the standard for data warehous integration.* John Wiley & Sons, 2002.

[PCTM03] John Poole, Dan Chang, Douglas Tolbert, and David Mellor. *Common Warehouse Metamodel – Developer's guide.* Wiley Publishing, 2003.

[Pos09] PostgreSQL Global Development Group. *PostgreSQL 8.4.2 Documentation*, 2009.

[PP03] Torsten Priebe and Günther Pernul. Ontology-based integration of olap and information retrieval. In *International Workshop on Database and Expert Systems Applications (DEXA)*, 2003.

[SKA+07] Il-Yeol Song, Ritu Khare, Yuan An, Suan Lee, Sang-Pil Kim, Jinho Kim, and Yang-Sae Moon. Samstar: A semi-automated lexical method for generating star schemas from an entity-relationship diagram. In *10th ACM International Workshop on Data Warehousing and OLAP (DOLAP 2007)*, 2007.

[SKA+08] Il-Yeol Song, Ritu Khare, Yuan An, Suan Lee, Sang-Pil Kim, Jinho Kim, and Yang-Sae Moon. Samstar: An automatic tool for generating star schemas from an entity-relationship diagram. *ER*, pages 522–523, 2008.

[SS91] Marc H. Scholl and H.-J. Schek. Supporting views in object-oriented databases. *Data Eng.*, 14(2):43–47, 1991.

[SSC08] Alkis Simitsis, Dimitrios Skoutas, and Malú Castellanos. Natural language reporting for ETL processes. In *Proceeding of the ACM 11th international workshop on Data warehousing and OLAP*, pages 65–72, Napa Valley, California, USA, 2008. ACM.

[Tha00] Bernhard Thalheim. *Entity-relationship modeling: foundations of database technology.* Springer Verlag, 2000.

[TLM03] J. Trujillo and S. Luján-Mora. A UML based approach for modeling ETL processes in data warehouses. *Lecture Notes in Computer Science*, pages 307–320, 2003.

[TVS07] Vasiliki Tziovara, Panos Vassiliadis, and Alkis Simitsis. Deciding the physical implementation of etl workflows. In *DOLAP '07: Proceedings of the ACM tenth international workshop on Data warehousing and OLAP*, pages 49–56, New York, NY, USA, 2007. ACM.

[Ver] Versant Corporation. *db4o 7.4 Reference Documentation*. `http://developer.db4o.com/Documentation/Reference/db4o-7.4/java/reference/`; visited on February 11, 2010.