JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

SOWI
Sozial- und Wirtschaftswissenschaftliche
Fakultät

An interpreter for a data definition and query language for hetero-homogeneous data warehouses

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften

(Mag.rer.soc.oec)

im Diplomstudium

Wirtschaftsinformatik

Eingereicht von:
Thomas Pecksteiner

Angefertigt am:
Institut für Wirtschaftsinformatik - Data & Knowledge Engineering

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

Mitwirkung:
Mag. Dr. Christoph Schütz

Linz, September 2015

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, September 2015

Thomas Pecksteiner

Abstract

Data warehouses enable companies to conduct advanced data analysis for supporting decision makers. Therefore, data are stored inside data warehouses from diverse, often heterogeneous operational data sources. The hetero-homogeneous data warehouse modeling approach allows for the integration of heterogeneous data. In this thesis we present SQL($\mathcal{M}$), a data query and manipulation language for working with hetero-homogeneous data warehouses. Furthermore, we describe the implementation of an interpreter for SQL($\mathcal{M}$) which reads, translates and executes SQL($\mathcal{M}$) input.

Kurzfassung

Data-Warehouses sind zentrale Sammelstellen für Daten und stellen Analysen und Berichte zur Verfügung. Firmen nutzen Data-Warehouses beispielsweise zur Unterstützung von Entscheidungsträgern. Zu diesem Zweck werden Data-Warehouses mit Daten aus diversen operativen Systemen befüllt. Diese Daten liegen oftmals in heterogener Form vor und werden im Standardfall in ein homogenes Schema überführt, wobei wichtige Informationen verloren gehen können. Der hetero-homogene Data-Warehouse-Ansatz ermöglicht es diese Heterogenitäten abzubilden und somit nicht zu verlieren. Diese Arbeit präsentiert SQL($\mathcal{M}$), eine Data Query und Manipulation Language für hetero-homogene Data-Warehouses. Darauf aufbauend werden Konzeption und Entwicklung des SQL($\mathcal{M}$) Interpreter veranschaulicht, dessen Aufgabe es ist SQL($\mathcal{M}$) -Anweisungen einzulesen, zu übersetzen und auszuführen.

Contents

List of Figures

List of Tables

Listings

1. Introduction

In today's business, information is a key factor for success. The purpose of a data warehouse is to represent information in an easily accessible way. Companies represent their business activities inside a data warehouse by integrating their operational data [CD97, p. 1 et seq.]. Therefore, data are extracted from different source systems, transformed and then loaded (see Section 2.2) into the data warehouse. The different operational data sources within a company often have heterogeneous schemata. The hetero-homogenous approach towards data warehouse modeling facilitates the integration of hetero-homogeneous data sources [NST10].

The data of interest that are stored in the data warehouse can be used for analysis to support the company's decision makers. Online analytical processing (OLAP) refers to tools and techniques for data analysis. The usability of a data warehouse system depends on many different elements. When it comes to OLAP an intuitive interface for working with the data is important. According to Codd it is one of the main criteria which define a good OLAP product [CD97, p. 12]. Hence, each data warehouse system has to solve the problem of how to provide the best possible usability to the user.

In this thesis we present an interpreter for SQL($\mathcal{M}$) , a query and manipulation language for working with hetero-homogeneous data warehouses. The SQL($\mathcal{M}$) language enables the analyst to access the hetero-homogeneous data warehouse in a more intuitive way than the PL/SQL API of the hetero-homogeneous data warehouse [Sch10]. The interpreter reads, translates and executes SQL($\mathcal{M}$) input using the PL/SQL API.

This thesis is organized as follows. Chapter 2 briefly presents underlying concepts and technologies. Chapter 3 revisits the hetero-homogeneous data warehouse modeling approach. Chapter 4 presents the main elements of SQL($\mathcal{M}$) and how to work with the language. Chapter 5 discusses interpreter design and implementation. Chapter 6 concludes the thesis. Appendix A describes the SQL($\mathcal{M}$) syntax. Appendix B presents the test plan for the implementation of the interpreter. Appendix C describes the installation process of the interpreter and the usage of the SQL($\mathcal{M}$) editor.

2. Background

This chapter gives an overview of the concepts and technologies needed for the construction of the SQL($\mathcal{M}$) interpreter. First, this chapter introduces the basics of compiler design, including language definition and formal grammars as well as the fundamentals of parsing. Second, this chapter explains the basic concepts of data warehousing. Finally, this chapter gives a brief introduction to object-relational database systems. Note that this chapter is not intended as an exhaustive overview but serves as a basic introduction to concepts needed for understanding this thesis. Thus, some details are omitted.

2.1. Compiler design

A compiler is a computer program which automatically reads and translates a source code into a target code. The whole transformation process is called compilation. The source code can be instructions specifying a computer program, a database query, or other. The compilation process requires a formal description of the valid structure of the source code. The definition of the structure is called syntax. The compiler analyses the validity of the source code based on the defined syntax and may construct an abstract syntax tree (AST), a representation of the source code independent of the concrete language, that serves as the basis for further semantic checks, optimization and translation. An interpreter does not, like a compiler, merely translate the source code but rather executes the source code on the target system, possibly using an intermediate representation obtained through compilation techniques [ALSU08, p. 3 et seq.]. A more thorough introduction to compiler design is provided by Wirth [Wir08] and Aho et al. [ALSU08].

There are different ways of designing the compilation process, namely multi-phase, single-phase and two-phase [Wir08, p. 2 et seq.]. The SQL($\mathcal{M}$) interpreter follows a two-phase compilation process, turning source SQL($\mathcal{M}$) code into PL/SQL code with subsequent execution of the generated code on the target system. The first phase, also called frontend or analysis, performs various checks and builds the AST. The second phase, also called backend or synthesis, performs the translation into the target code.

Figure 2.1 illustrates the sequence of compilation phases. The compilation process can be separated into two sub-processes, which are analysis and synthesis. The structure of the analysis process can be summarized as follows [Pun09, p. 2 et seqq.], [Wir08, p. 2 et seqq.], [ALSU08, p. 6 et seqq.]:

- *Lexical analysis* is the first phase of the compilation and is also called scanning. In this phase the source code is read and structured based on the grammar of the source code and then transformed into a sequence of tokens. During this transformation from source code to tokens, the lexer also clears the input stream from symbols that are specified to be ignored, e.g. blanks or tabs. Further, the lexer can specify error messages that occur during the compilation process.

- *Syntax analysis* is the second phase and is also called parsing. The sequence of tokens generated during the lexical analysis is the input for the parsing process.

The parser checks if the token sequence is valid with respect to the source grammar. Syntax analysis may also create an abstract syntax tree out of the provided tokens.

- *Semantic analysis* is the third phase which may use the abstract syntax tree to perform semantic checks over the source code. For example, type checks may be performed during this phase.

The structure of the synthesis process, which succeeds the analysis process, can be summarized as follows [Pun09, p. 4 et seqq.], [ALSU08, p. 13 et seqq.]:

- *Code optimization* may reorganize the abstract syntax tree in order to improve performance of the target code.
- *Code generation* generates target code from the abstract syntax tree.

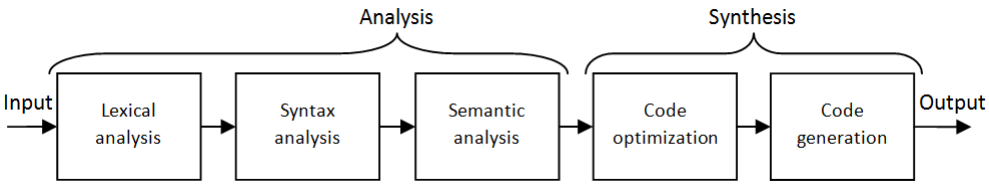


Figure 2.1.: The phases of a compiler (based on [Pun09, p. 2 et seq.])

The grammar defines the valid syntax of the source code. The Extended Backus Naur Form (EBNF, [Wir77]) is a formal notation for grammars. The JavaCC notation is an EBNF variant. Consider, for example, the grammar in Listing 2.1.

```
SelectStatement : "SELECT" SelectClause "FROM" FromClause
SelectClause : "*" | ColumnList
ColumnList : ColumnIdentifier ("," ColumnIdentifier )*
FromClause : TableName (Alias)
ColumnIdentifier : ( Alias "." ) ColumnName
```

Listing 2.1: EBNF example in simplified JavaCC notation

The example grammar in Listing 2.1 defines a structure for simple database query statements; it illustrates the four elements of grammars, namely terminal symbols, non-terminal symbols, start symbol and production rules [ALSU08, p. 239]. The example grammar consists of five production rules which specify how the non-terminal symbols *SelectStatement*, *SelectClause*, *ColumnList*, *FromClause* and *ColumnIdentifier* are replaced. The production rules define the composition of non-terminal symbols through other non-terminal and terminal symbols. Terminal symbols cannot be further replaced by other symbols. Double quotes mark terminal symbols, for example "SELECT". Vertical bars signify choice. Brackets signify optional symbols. An asterisk signifies repetition. In the example the start symbol is the *SelectStatement*.

A parser implements the syntax analysis phase of the compilation process. Different types of parsers exist, namely universal parsers, top-down parsers and bottom-up parsers [ALSU08, p. 233 et seqq.]. A universal parser is able to handle any type of grammar but is rather inefficient. Top-down and bottom-up parsers are the commonly used variants. The SQL($\mathcal{M}$) interpreter follows a top-down parsing approach.

Reading direction and lookahead effort are other distinguishing features of parsers. The SQL($\mathcal{M}$) interpreter follows an LL(k) parsing approach, meaning that the parser reads the input from left to right, syntax analysis is performed top-down or derived from left, with a *lookahead* of k tokens. In the case of the SQL($\mathcal{M}$) interpreter, k generally equals one, meaning that the parser only needs the next expression to determine the substitution of a production rule. However, individual parts of the SQL($\mathcal{M}$) grammar require a local lookahead greater than one. Consider, for example, the grammar for simple database queries in Listing 2.1. Assuming that the symbols *Alias* and *ColumnName* of the example grammar match to some character sequence, most of the grammar is defined as LL(1), with the exception of production rule *ColumnIdentifier*. The production rule *ColumnIdentifier* is substituted by an optional *Alias* followed by an obligatory *ColumnName*. For the parser to be able to decide if the production rule applies with the optional *Alias* or not, it needs to find the discriminating point and therefore requires the next two expressions. Therefore, either the parser needs to be LL(2) or the grammar may specify a local lookahead of two for the production rule *ColumnIdentifier*. In many cases a rewriting of the grammar can also reduce the number of a local lookahead greater than one.

A parser generator is a tool that helps to build the frontend of a compiler. It automatically generates a syntax analyser according to a grammar description of the source language. Examples of available parser generators are Coco/R[1], JavaCC[2] and ANTLR[3].

2.2. Data warehousing

A data warehouse is a collection of data for decision . A company often stores data in many different operational systems. Analysis of the company data requires the integration of the various operational data stores. A data warehouse combines the company A data warehouse contains the company that is important for decision making and supports the user with analysis functionality. According to Inmon a data warehouse is characterized as "a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions" [Inm02, p. 29].

Subject-oriented means that the data are structured by the subject areas of the respective company. Typical subjects are customers, products or locations. As already mentioned, a data warehouse usually **integrates** multiple heterogeneous data sources. Thereby the data is homogenized so that all data inside the data warehouse satisfies the used data model. Data warehouses are further characterized as **nonvolatile** which distinguishes them from transactional databases. The data inside a data warehouse changes rarely. Usually new information does not replace the old one but is stored as additional data. With the integration of historical data a data warehouse is able to provide **time-variant** analysis [EN07, p. 979]. This is possible since each data item inside the data warehouse has some sort of timestamp to indicate the time period that the data item is valid for [Inm02, p. 34].

The term Online Analytical Processing (OLAP) refers to the analysis of data inside a data warehouse. Sometimes the term OLAP is even used synonymous to the term data warehousing. The purpose of OLAP is the opposite to online transactional processing (OLTP). In general, OLTP refers to the task of operational database systems to frequently

[1] http://www.ssw.uni-linz.ac.at/Coco/
[2] https://javacc.java.net/
[3] http://www.antlr.org/

create, update an delete data. In contrast data warehouses are not transaction-driven but are specialized in analysing huge amounts of data [EN07, p. 977 et seq.].

According to Codd data warehouses, respectively OLAP processes, should fulfill the following characteristics [Cod93, p. 12]:

- Multidimensional conceptual view
- Generic dimensionality
- Unlimeted dimensions and aggregation levels
- Unrestricted cross-dimensional operations
- Dynamic sparse matrix handling
- Client/Server architecture
- Multi-user support
- Accessability
- Transparency
- Intuitive data manipulation
- Consistent reporting performance
- Flexible reporting

Figure 2.2 illustrates the main activities of data warehousing. First, the extract, transform and load (ETL) process integrates the available data from the different operational systems. During the transformation process the homogeneities inside the data usually are eliminated. After loading data into the data warehouse, tools like decision support systems (DSS), executive information systems (EIS) and OLAP clients can start to analyse the data inside the data warehouse, to generate information for decision support [EN07, p. 977 et seq.].

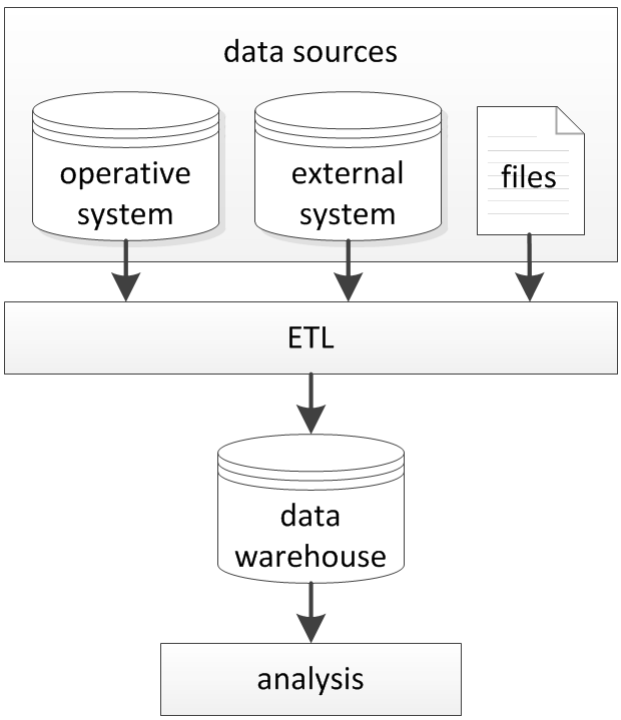


Figure 2.2.: The conceptual structure of a data warehouse (based upon [FAR11, p. 9])

Data warehouses employ multidimensional data models. The multidimensional data model organizes facts in an n-dimensional space, also called data cube [JLVV03, p. 87

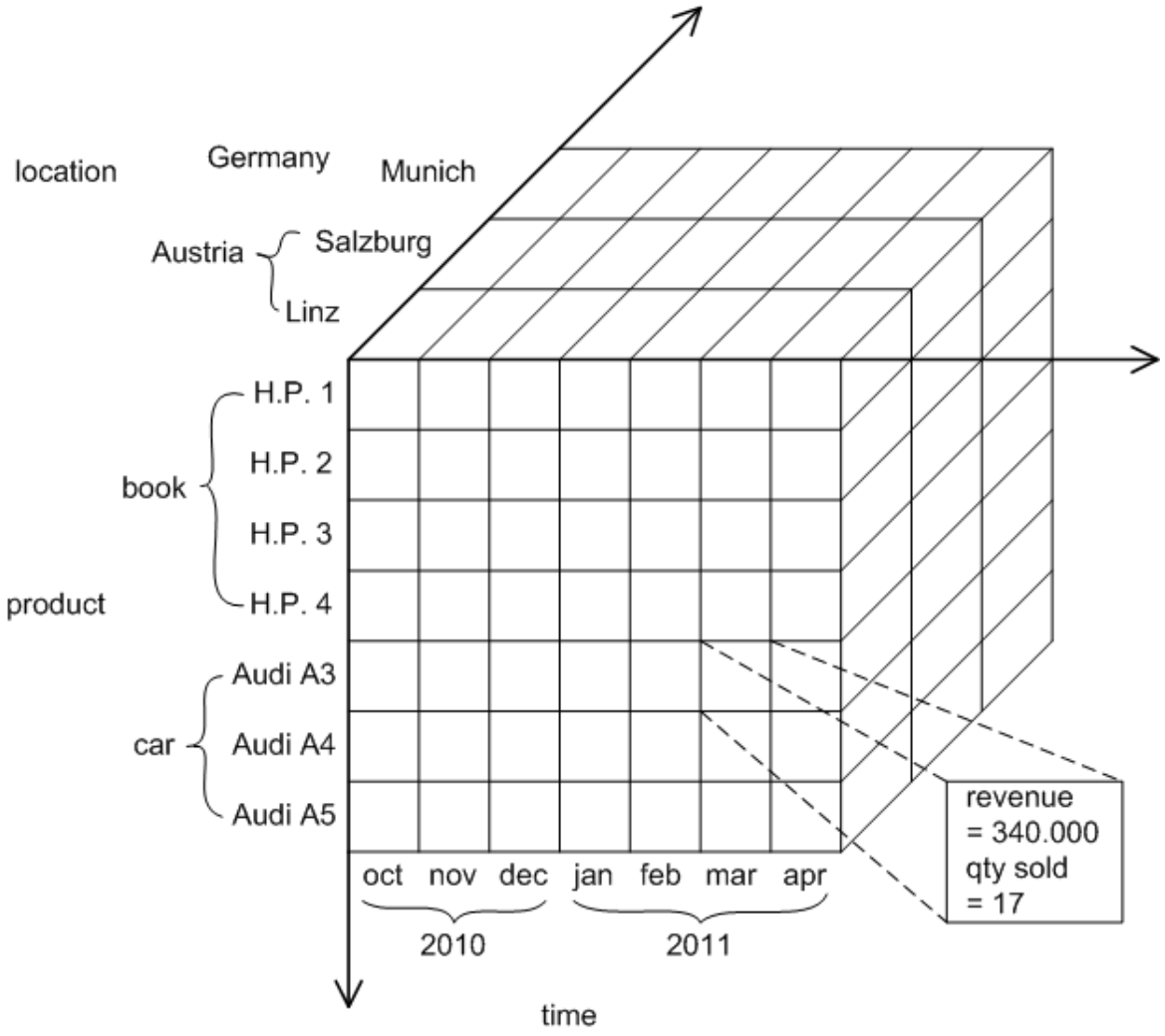


Figure 2.3.: A three-dimensional sales cube

et seq.]. Figure 2.3 shows a cube that consists of three dimensions, namely *Product*, *Time* and *Location*. Each dimension of a cube consists of dimension and non-dimension attributes [GMR98]. Dimension attributes are also called levels and are organized in hierarchies. Non-dimension attributes, or simply attributes, further describe the levels.

Hierarchies have an implicit *all* level, also called *top*. The *all* level represents the most abstract level in a hierarchy. In Figure 2.3, the hierarchy of the *time* dimension consists of the levels *month*, *year* and *all*, where *month* rolls up to *year* and *year* rolls up to the *all* level. The structure of a dimension hierarchy is influenced by the requirements of the analysis [JLVV03, p. 87 et seq.], [EN07, p. 980 et seqq.].

Next to dimensions a cube also contains measures. Measures, for example *revenue* or *qty sold* in Figure 2.3, quantify the facts represented by the coordinates in the multidimensional space. Figure 2.3 shows such a coordinate at the point (Audi A3, March 2011, Linz). In this point the measure *revenue* has the value 340,000 and *qty sold* has the value 17 for Audi A3s sold in March 2011 in the city Linz. Depending on the selected levels at each dimension, the user receives a single point inside the cube, a slice of the cube, a sub-cube or the entire cube with all data.

The measures are aggregated along the dimension hierarchies [JLVV03, p. 87 et seq.]. The following operations are typically available [JLVV03, p. 90 et seq.], [EN07, p. 980 et seq.]:

- *Roll-up* refers to the aggregation of data. The rollup operation groups the data by the levels of the cube dimensions.
- *Roll-down, or Drill-down,* changes the level of granularity to be more finely grained.
- *Slice and dice* refer to the selection of a subset of a cube by specifying dice coordinates or conditions over the non-dimension attributes.

Building the data warehouse, usually requires the integration of data from several different operational data sources. Besides the extraction of data, the ETL process further consists of the transformation of the extracted heterogeneous data, to fit the multidimensional data warehouse schema. Usually this step also involves cleaning of the data. As the last step follows the loading of the adapted data into the data warehouse [CD97, p. 3]. The SQL($\mathcal{M}$) language provides statements for loading data into the data warehouse.

Relational databases may store multidimensional models. Working with data warehouses that build on multidimensional models is also referred to as relational OLAP (ROLAP). The star schema organization of tables is a common logical representation for the multidimensional model. The star schema consists of a fact table which references multiple dimension tables [PS99] [MK00, p. 3 et seq.]. Some database vendors (for example Oracle [Odw07, 20-1 et seqq.]) extend the SQL standard to provide additional functionality for performing ROLAP.

2.3. Object-relational databases

Object-relational databases extend the traditional relational model with object types (see [Oor08, 1-1 et seq.]). Object types define attributes, functions and procedures. Object tables are collections of objects that conform to the same object type. Functions and procedures encapsulate query semantics and manipulation operations. To implement the behavior of objects the Oracle database system provides PL/SQL, the procedural extension of SQL, which developers may also use to define stored procedures. Oracle also provides the possibility to define Java stored procedures [Ora09, 5-1 et seqq.]. The SQL($\mathcal{M}$) interpreter runs as a Java stored procedure and uses a PL/SQL API for hetero-homogeneous data warehouses stored in an object-relational database.

Client applications often access object-relational database systems using dynamic SQL. The Java Database Connectivity (JDBC) serves as the API that allows the use of dynamic SQL in Java applications (see [Ojd08, 1-1 et seq.]). Thus, applications hold SQL statements in string variables, which are sent to the database server, and receive sets of tuples as result, which are then processed iteratively. JDBC allows for the calling of stored procedures and the execution of dynamic PL/SQL blocks.

3. Hetero-homogeneous data warehouses

In the following sections an introduction is given about the underlying concepts and software products, on which the presented work is based on. First the concept of hetero-homogeneous hierarchies is presented. Thereby it will be shown why this concept is needed and how the modeling process looks like. After that, a data warehouse prototype will be introduced, implementing the hetero-homogeneous concept.

3.1. Hetero-homogeneous dimension hierarchies and cubes

The common approach to implementing a data warehouse is to define a global homogeneous schema as basis [CD97, p. 2]. Thus originates the problem that data (business related or scientific), which often exists in heterogeneous form, must be converted to suffice the homogeneous schemata dictated by the data warehouse. To achieve a valid conversion, often heterogeneous parts of the data pool must be dropped. This happens during the transformation phase of the ETL process (see Section 2.2). However, the eliminated data could contain important information for analysis and as a consequence could influence the actions of decision makers. So ways of integrating this heterogeneous information are needed. The approach of hetero-homogeneous dimension hierarchies provides such an integration.

A hetero-homogeneous dimension hierarchy is built of *multilevel objects*. The concept of m-objects and the related multilevel modeling approach was first introduced in [NGS09]. In [NST10] the approach of m-objects has been extended and introduced for modeling hetero-homogeneous data warehouses. The general concept of modeling techniques for multilevel abstraction is discussed in [NST11].

The idea behind m-objects is to describe real-world entities at several abstraction levels. This sequence of abstraction levels is called level hierarchy. The different levels of a level hierarchy are ordered from the most abstract to the most specific one. The most abstract level of a level hierarchy is called the top level. Every m-object can concretize the level hierarchy beneath itself by including new abstraction levels. Thereby it is situated at the top level of its own level hierarchy. Through the level hierarchy a m-object not only defines itself, but also all common properties of m-objects at more concrete levels. At each level inside the level hierarchy non-dimensional attributes can be defined to create a data model.

As a consequence, different level hierarchies can exist inside a dimension hierarchy. One homogeneous global level hierarchy and heterogeneous local sub level hierarchies. The global level hierarchy is defined by the top m-object situated at the most abstract level, i.e. the top level of the whole dimension hierarchy called *ALL* or *TOP*. It is the minimal schema, which each sub-level hierarchy, defined by any m-object in a concrete

relationship with the start m-object, must be consistent with. This ensures a certain degree of homogeneity over the whole dimension hierarchy. But each m-object can specify its own heterogeneous schema starting from its abstraction level. So sub level hierarchies can contain various additional levels, for better describing the local m-objects. By using the hetero-homogeneous dimension concept as introduced, one receives a schema, that is as homogeneous as possible and additionally as heterogeneous as needed, for describing real world entities as m-objects.

As already mentioned, m-objects can be further described by defining attributes to represent non-dimensional properties of m-objects at a certain level. Each attribute is thereby only defined for one individual abstraction level and thus only provides values for m-objects being located at this level. Just like the extension of level hierarchies, also the number of attributes can differ in different sub level hierarchies. Each m-object can introduce new attributes at abstraction levels located inside its level hierarchy.

The following example will demonstrate the usage of m-objects and the advantages of hetero-homogeneous dimension hierarchies. As a starting point a homogeneous *sales*-cube modeled with the Dimensional Fact Model (DFM) is given (see Figure 3.1).

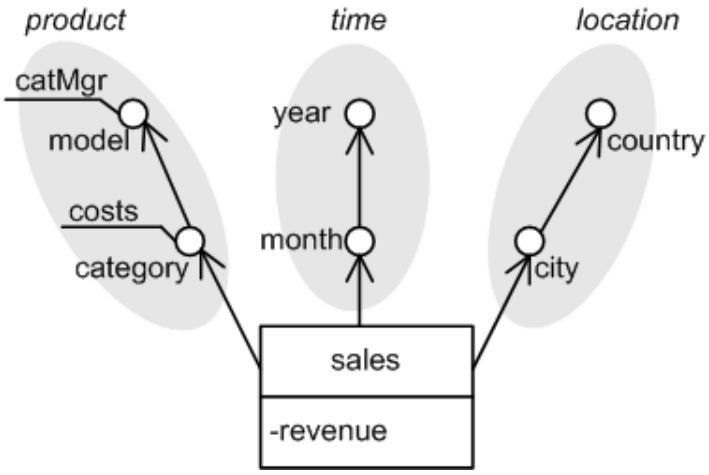


Figure 3.1.: Homogeneous cube modeled with the Dimensional Fact Model
based on Figure 1 from [NST10, p. 1]

The illustrated cube from Figure 3.1 consists of the dimensions *Product*, *Time* and *Location*. Important for now is only the dimension *Product*. It consists of the levels *category* and *model*, and the non-dimensional attributes *costs* and *catMgr*.

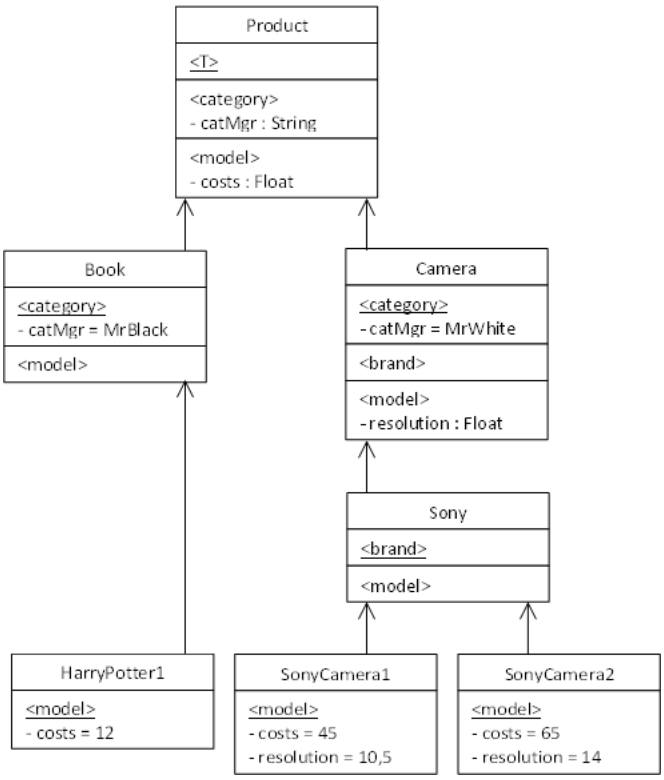


Figure 3.2.: The hetero-homogeneous product dimension of a sales cube modeled with m-objects based on Figure 2 from [NST10, p. 3]

Figure 3.2 shows how m-objects can be used to model the *product* dimension of Figure 3.1 as a dimension hierarchy. The pictured hetero-homogeneous dimension contains the root m-object *Product*. The whole dimension hierarchy is homogeneous regarding the level hierarchy defined by the root m-object, which is:

$$top \} category \} model$$

This level hierarchy represents the minimal schema and must be valid through out the whole dimension hierarchy. But as already mentioned, this schema can be extended. The m-object *Camera*, specializes the level hierarchy by introducing an additional level *brand*. This extension has no influence on the other sub dimension hierarchies starting at m-object *Book*. There the level structure remains as defined by the root m-object *Product*. Thereby the dimension hierarchy is heterogeneous due to the fact that the level hierarchies of different sub-dimension hierarchies are not equivalent. Not only a level-hierarchy is extended, but also the attributes describing the m-objects. The *Camera* m-object extends the level *model* with a second attribute called *resolution*, for a more precise description of cameras. This change only influences m-objects at the *model* level located in the sub-dimension hierarchy starting at the m-object *Camera*. M-objects of other sub-dimension hierarchies at the *model* level are not affected.

Cubes provide the functionality needed for analysis on the stored data inside a data warehouse. The hetero-homogeneous concept provides multilevel cubes, in short m-cube. Each m-cube consists of multiple dimensions. For each dimension of the m-cube a m-object must be declared. This m-object represents the limit of abstraction inside the

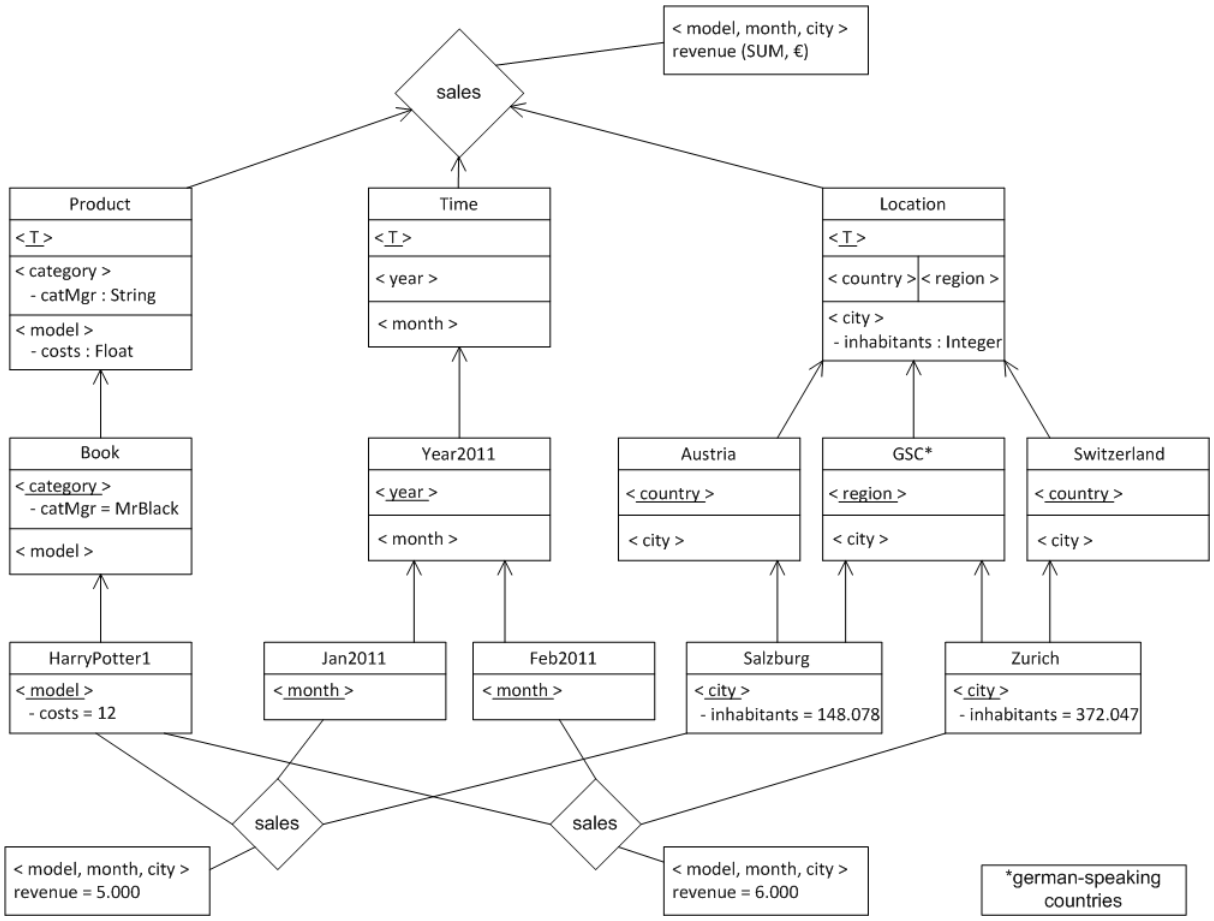


Figure 3.3.: A hetero-homogeneous sales cube modeled with m-objects based on Figure 4 of [NST10, p. 5]

m-cube for the respective dimension hierarchy. This sequence of m-objects and their corresponding dimension hierarchies is called the m-cube's *root-coordinate*.

Figure 3.3 shows the m-cube *sales*. It is defined by the root-coordinate <Product IN product_dim, Time IN time_dim, Location IN location_dim>.

Data in form of measures, and their assigned values, are defined by so called multilevel facts. M-facts describe relationships of m-objects at different levels of abstraction, just the same way as m-objects describe real-world entities. A m-fact is defined by a sequence of m-objects, called coordinate, which represents the position of the m-fact inside a cube. Thereby the m-fact must be consistent with the m-cube it is defined for. This means that the coordinate of a m-fact must match with the data structure of the m-cube, defined through the m-cube's root-coordinate. Looking at the previously defined *sales* cube's root-coordinate, this implies that every m-fact for this m-cube must be constructed of one m-object from the dimension product_dim, one m-object from the dimension time_dim and one m-object from the dimension location_dim. Further the m-objects Product, Time and Location represent the abstraction limit. Therefore m-objects situated on a higher abstraction level as those three m-objects, inside the respective dimension hierarchies, are not allowed. Measures of m-facts consist of a name, an aggregation function, a unit declaration, a value and a connection level. The connection level of a m-fact defines the granularity of a measure. M-facts defined at the connection level of an introduced measure, serve as the data basis for the measures of m-facts with lower granularity. This

means that the values of measures of a m-fact on a lower abstraction level are summed up to generate the values for measures of m-facts on a higher abstraction level.

As an example, look at the m-facts in Figure 3.3. The m-fact <Product, Time, Location>, which represents the relationship between this three m-objects, is the most abstract m-fact inside the m-cube *sales*. It introduces the measure *revenue*. Besides the declaration of the measure name and value unit, also the aggregation function and the aggregation basis, the connection level, is defined. The connection level determines from where the values for a measure are aggregated and the aggregation function defines how they are calculated. The measure *revenue* defines the aggregation function SUM (summing up) and the connection level <model, month, city>. This implies that the measure *revenue* aggregates the values from all m-facts, where the connection level is consistent with the levels <model, month, city>. The m-fact <HarryPotter1, Jan2011, Salzburg> is such a m-fact, as it fulfils the conditions of the connection level of the measure revenue. Therefore the value of the measure *revenue* of the m-fact <HarryPotter1, Jan2011, Salzburg> is a part of the value of the measure *revenue* of the m-fact <Product, Time, Location>. This way all measure values are calculated to result in more abstract measure values. A detailed description of how m-facts can be declared and what aggregation options are possible is given in Chapter 4.

3.2. Hetero-homogeneous data warehouse prototype

Based on the approach of hetero-homogeneous dimension hierarchies and cubes a data warehouse prototype was developed by extending an Oracle 11g database software. The prototype was introduced and described by Schütz [Sch10]. The latest prototype is available at [HH-DW]. The following section gives an overview over the prototype used for the implementation of the SQL($\mathcal{M}$) interpreter.

The hetero-homogeneous data warehouse was implemented with the use of Oracle's object-relational features, including object types, functions and procedures to create complex application code. This implies the definition of multidimensional structures through m-objects and dimension hierarchies and the creation of m-cubes, based on the defined dimensions. After the m-cube creation, m-facts and corresponding measures are inserted for providing data. The query functionality grants access to the stored data inside the data warehouse on multiple abstraction levels.

3.2.1. Defining dimension hierarchies and cubes

There exist two fundamental structures inside the hetero-homogeneous data warehouse, dimension hierarchies and multilevel cubes. The first step of building a data warehouse, is defining its dimensional structure, starting with dimension hierarchies. After the creation of a dimension, m-objects can be defined for it. During the creation of a m-object new levels can be introduced by extending the level hierarchy. They can not be added afterwards. On the other hand, attributes can be defined at each point in time for a m-object.

For defining m-cubes a list of dimensions has to be assigned, which represent the different of the cube. Any number of dimensions is thereby valid. Further a m-object for each dimension is needed for setting the entry point. Measures can be defined at the m-cube creation or at any point later in time. After a m-cube is created m-facts are defined to populate the measures of a m-cube with values. M-facts consists of a cube reference and

a coordinate. The coordinate defines where the m-fact is situated inside the referenced m-cube.

For certain data definition and manipulation processes, so called bulk functionality is available. Bulk methods combine individual statements to optimize the performance of the data warehouse. For following processes bulk methods exist:

bulk type	**method name**	**preconditions**
creating m-objects	bulk_create_mobject	same parent m-objects and identical level hierarchy
setting attribute values	bulk_set_attribute	identical attribute inside a dimension hierarchy
creating m-facts	bulk_create_mrel	same m-cube
setting measure data	bulk_set_measure	same m-cube and equal measure name

Table 3.1.: Types of BULK statements

Table 3.1 not only shows the different bulk statement types but also the needed preconditions to merge individual statements. The advantage of bulks statements deduce from the reduced effort for preconditions checks, consistency checks and other validations as they have to be done only once for a bulk method, and not for each individual statement. Further the implementation of the different bulk methods, uses the FORALL loop, which grants further process speed enhancements.

For providing all of the described functionality the object-relational features of the Oracle database are used. The different kind of hetero-homogeneous entities (dimensions, m-objects, m-cubes, m-facts, etc.) are implemented as separate object types, which are dynamically generated at runtime, representing the corresponding entity. For the different entities basic object types exists which serve as templates. They define the general design and functionality of the different object types. All dynamically generated types derive from these basic object types and concretize them. For example the object type mobject_ty is the basic object type for multilevel objects. For every dimension a separate mobject_ty object type is generated and named like mobject_*_ty, the asterisk stands for the unique identifier of the corresponding dimension hierarchy and is inserted into the name string during the creation of the object type.

After one of the described objects is generated, certain properties can no longer be modified. E.g. after a m-cube is defined, the number of dimensions it consists off can not be altered anymore. However, one and the same dimension can be used in different m-cubes at the same time. If a dimension hierarchy is altered, affected m-cubes have automatically access to the new data. Therefore the modeling of the hetero-homogeneous schema is very important, since subsequently appearing problems most often can not be fixed afterwards. In such cases the data warehouse has to be designed from scratch all over again.

3.2.2. Querying multilevel cubes

The allocation of data is the main task of a data warehouse and is done through querying the measure data stored inside cubes. The hetero-homogeneous data warehouse prototype provides *closed m-cube queries* for the data inquiry. The result of a query is calculated by using queryviews. Queryviews are object types providing the needed calculation functionality. The data warehouse prototype generates for each m-cube an object of queryview_*_ty type. A queryview object is dynamically defined at the same time a m-cube is generated. The * represents the unique identifier of the m-cube, for which the queryview object is designated. Beside a m-cube reference a queryview contains a list for storing query expressions. The three provided operations are:

- *dice*
- *projection*
- *slice*

All three operations process m-cubes and return a result m-cube with the respective changes. Thereby the *dice* functionality reduces a m-cube by reducing the abstraction level of the cube's root coordinate. The *projection* operation allows to exclude measures, so that only the needed measures are retrieved. With *slice* expressions criteria can be defined which have to be fulfilled by the m-facts of the queried m-cube for being used during the calculation process.

With these three operations the data inside a m-cube can be reduced to any needed granularity. The expressions are realized through the object type expr_*_ty. This type serves as the basic object type. Each operation has its own specific realization, called dice_expr_*_ty, project_expr_*_ty and slice_expr_*_ty. Here too the object types are created dynamically for one specific queryview.

For accessing the hetero-homogeneous data warehouse different options are available. Besides the data definition and querying language SQL($\mathcal{M}$) described in this thesis, a hetero-homogeneous OLAP client exists [Rot15], which extends there the open source OLAP engine Mondrian for supporting the hetero-homogeneous dimension hierarchies concept.

4. SQL($\mathcal{M}$): A data definition and query language

SQL($\mathcal{M}$) is a data definition and query language designed for an end-user friendly access to hetero-homogenous data warehouses. The name SQL($\mathcal{M}$) was chosen as reference to well known SQL. Certain function names were adopted from SQL, as well as their underlying functionality, but were extended for a multilevel purpose. E.g. the command *CREATE* means the same in SQL($\mathcal{M}$) as in SQL. This was done because of usability issues, to ensure that users will have a low period of adjustment and training. Also some standard SQL code is integrated into SQL($\mathcal{M}$) to further enlarge the functionality of the language, as the SQL($\mathcal{M}$) query functionality is integrated into the *SQL SELECT statement*.

In general SQL($\mathcal{M}$) statements can be sub-divided into the following categories:

- data definition (DDL): statements which are used for defining the data warehouse schemata. (CREATE, ALTER and DROP)
- data manipulation (DML): statements which set or update values (UPDATE)
- data query (DQL): statements which are used for querying data from the data warehouse (SELECT)
- optimization (BULK): statements which combine arbitrary DDL or DML statements into one single statement

In the following sections detailed information for all available SQL($\mathcal{M}$) statements is given. They are presented by category as displayed above. For a better understanding, for all statement types examples are presented. They represent the data warehouse structure shown in Figure 4.1. For a more defined definition of the SQL($\mathcal{M}$) grammar, look at Appendix A, where statements are visualized as railroad diagrams.

Figure 4.1 presents the data warehouse structure for the following statement examples. It shows a m-cube (*sales_cube*) consisting of three dimensions (*product_dim*, *time_dim*, *location_dim*). The m-cube introduces one measure called *revenue*. Also the m-facts for this example are visualized inside the table located in Figure 4.1.

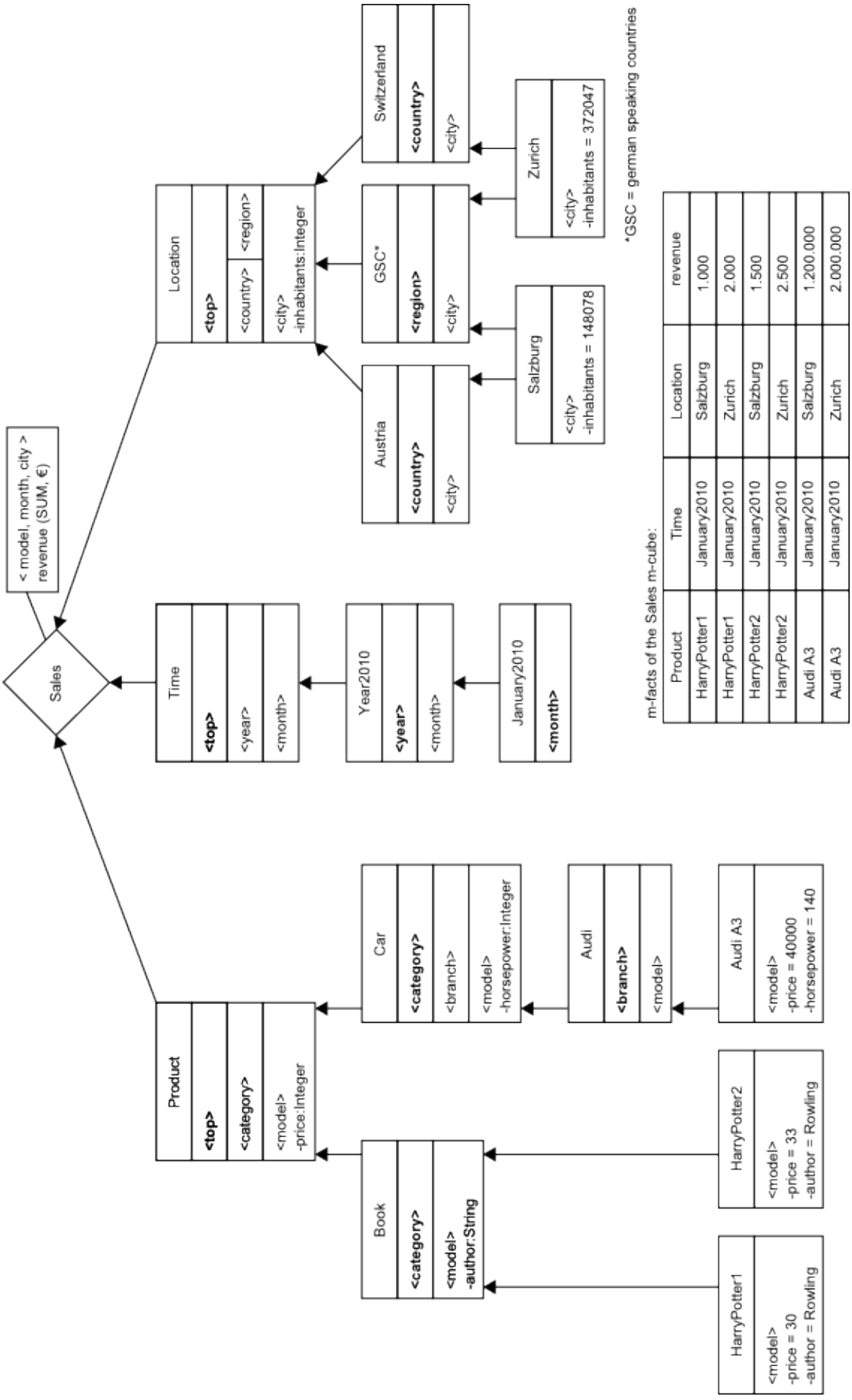


Figure 4.1.: Data warehouse model used for statement examples

4.1. Data definition

With the use of data definition statements the structure of hetero-homogenous data warehouses is defined. These statements can be divided into three different types, which are CREATE, ALTER and DROP. Following, all available data definition statements will be presented, sorted by their types. Every statement type is introduced by illustrating the important information, such as necessary preconditions and so forth. Also examples are given for all statements, to illustrate their structure and usage in detail.

Create dimension hierarchy

Dimension hierarchies are the foundation of hetero-homogeneous data warehouses. They need to be created first when building a data warehouse structure. For creating a dimension hierarchy only a name for the dimension is needed. No further optional input is possible. For a dimension name the unique name assumption must be satisfied. This means that two dimension hierarchies cannot have the same name. Listing 4.1 shows an example of a CreateDimensionHierarchy statement.

Create multilevel object

Multilevel objects are used to build up dimension hierarchies. For their creation some mandatory data is needed:

- multilevel object name
- dimension hierarchy name
- abstraction level name

M-object names have to fulfill the unique name assumption, just like dimension names. But in this case the name must only be unique inside the corresponding dimension hierarchy, which implies that m-objects situated inside different dimension hierarchies can have the same name.

After the declaration of the mandatory data, multiple optional input is possible. A m-object can be specified with or without declaring a parent m-object. When no parents are defined the m-object must be at the top level of the dimension hierarchy. Whether a m-object is at the top level or not, has consequences for the definition of the level hierarchy. Top level m-objects (root m-objects) **must** define a level hierarchy, as they introduce the homogeneous part of the hierarchy, which must be consistent over the whole dimension. Thereby the first level introduced is the top-level. Non-root m-objects inherit the dimension hierarchy from the parent m-object. Further the user can declare additional abstraction levels (heterogeneous parts) for non-root m-objects. When extending the level hierarchy, not the whole level hierarchy must be specified. The explicit mentioning of the affected levels alone is sufficient, as the SQL($\mathcal{M}$) interpreter autocompletes missing parts of a level hierarchy at run time. Affected levels are those levels that in the inherited level hierarchy roll up to the newly introduced level's parent. It is important to know that new levels can only be added during the creation of the m-object, and not afterwards. Besides new levels also new attributes can be added. This is not only possible during the creation of a m-object, but they can also be added later, by using an ALTER statement. To add an attribute a name and the data type (e.g. NUMBER) of the attribute must be

declared and the level where it should be situated. At the end of the statement, values can be set for already introduced attributes. This is optional, as attribute values can be set and changed later on with the help of UPDATE statements.

Listings 4.1 to 4.4 present three examples of CreateMultilevelObject statements, showing different variations of this statement type. Thereby Listing 4.1 displays a root level m-object. It introduces the first level hierarchy, consisting of three abstraction levels. The level top is thereby the most abstract level of this level hierarchy, defined with the command CHILD OF NULL.

```
1 CREATE DIMENSION HIERARCHY product_dim;
2
3 CREATE MULTILEVEL OBJECT Product IN DIMENSION product\_dim AT LEVEL top
4 HIERARCHY (
5    LEVEL top CHILD OF NULL,
6    LEVEL category CHILD OF top,
7    LEVEL model CHILD OF category
8 );
```

Listing 4.1: Create Dimension Hierarchy statement

Listing 4.2 illustrates a m-object introducing attributes at abstraction level *category*. The two attributes contain two different data types, namely VARCHAR2 and NUMBER. Inside the braces of data type VARCHAR2 the maximum length of the string is determined, as in the SQL standard.

```
1 CREATE MULTILEVEL OBJECT Book IN DIMENSION product\_dim AT LEVEL category
     UNDER Product
2 HIERARCHY (
3    LEVEL category CHILD OF top,
4    LEVEL model CHILD OF category (
5       author VARCHAR2(40),
6       price NUMBER
7    )
8 );
```

Listing 4.2: Basic Create Multilevel Object statement

The next statement, displayed in Listing 4.3, shows how an additional level is introduced. The indroduction of a new level can be done by defining the entire new level hierarchy with both additional levels and inherited levels. Alternatively, it would also be sufficient to only explicitly mention the newly introduced levels and all child levels that are affected by the introduction. In this example the level *branch* is introduced during the creation of the m-object *Car*. This is done by defining the newly added level *branch*. Additionally, the level *model*, which is a child of the newly added level's parent, must now be defined as a child of *branch*.

```
CREATE MULTILEVEL OBJECT Car IN DIMENSION product\_dim AT LEVEL category UNDER
    Product
HIERARCHY (
   LEVEL branch CHILD OF category,
   LEVEL model CHILD OF branch (
      horsepower NUMBER
   )
);
```

Listing 4.3: Create Multilevel Object statement introducing an additional level

Listing 4.4 shows how to set values of attributes at m-object creation. The two m-objects (HarryPotter1 and HarryPotter2) inherit their level hierarchy and associated attributes as defined by its parent m-object. Further, values for the inherited attributes are set.

```
CREATE MULTILEVEL OBJECT HarryPotter1 IN DIMENSION product\_dim AT LEVEL model
    UNDER Book
SET price = 30,
    author = 'Rowling';
CREATE MULTILEVEL OBJECT HarryPotter2 IN DIMENSION product\_dim AT LEVEL model
    UNDER Book
SET price = 33,
    author = 'Rowling';
```

Listing 4.4: Create Multilevel Object statement with attribute assignment

The statements in Listing 4.5 show how multiple inheritance between m-objects can be defined. The m-objects at the level city (Salzburg and Zurich) are associated with the level *country* as well with the level *region*. They would inherit specifications from both. E.g if an attribute *canton* would be added at m-object *Switzerland* at level *city* the m-object *Zurich* would inherit this attribute. M-objects created under *Austria* at level city (e.g. Salzburg) would not inherit it. Vice versa an attribute added for *GSC* (German speaking countries) at level *city* would concern both m-objects at this level, meaning *Zurich* as well as *Salzburg*.

```
CREATE MULTILEVEL OBJECT Location IN DIMENSION location\_dim AT LEVEL top
HIERARCHY( LEVEL top CHILD OF NULL,
               LEVEL country CHILD OF top,
               LEVEL region CHILD OF top,
               LEVEL model CHILD OF (country, region) (
                      inhabitants NUMBER
               )
);
CREATE MULTILEVEL OBJECT Salzburg IN DIMENSION location_dim AT LEVEL city
    UNDER Austria, GSC
    SET inhabitants = 148078;
```

Listing 4.5: Create Multilevel Object statement with multiple inheritance

Create multilevel cube

Multilevel cubes consist of arbitrary dimension hierarchies, stated as root coordinate of the m-cube. Beside the root coordinate a m-cube further needs a unique name amongst all cubes. Listing 4.6 shows how to create a m-cube consisting of three dimensions. Basically an arbitrary number of dimension can be assigned to a cube, but at least one. For each dimension an associated m-object is defined, which constitutes the most abstract level of the dimension inside the m-cube.

```
CREATE MULTILEVEL CUBE sales_cube OF DIMENSIONS
   Product IN DIMENSION product_dim,
   Time IN DIMENSION time_dim,
   Location IN DIMENSION location_dim;
```

Listing 4.6: Create Multilevel Cube statement

Create multilevel fact

CreateMultilevelFact statements define schema and base data of the hetero-homogeneous data warehouse. To create m-facts two things are mandatory. First, the multilevel fact's coordinate in the cube. Second, the multilevel cube name where the m-fact is located in.

The m-fact's cube coordinate defines the position of the m-fact inside the cube. The m-fact's cube coordinate can be declared in two different ways, namely qualified and unqualified. A qualified coordinate contains, contrary to an unqualified coordinate, not only the m-object names but also the name of the dimension they belong to. The benefit of a qualified coordinate is that it can be validated before the statement is processed, and if the order of the m-objects inside the m-fact cube coordinate does not correlate with the order of the m-cube coordinate, the divergence will be repaired by the SQL($\mathcal{M}$) interpreter if possible. By using an unqualified m-fact cube coordinate the list of m-objects will be taken as given. The order of the m-objects inside the m-fact cube coordinate must in this case be the same as the order defined by the corresponding m-cube coordinate. Hereby, the user is responsible for the validity of their input. After the m-fact creation, measures can be defined. Therefore, different inputs are needed for the measure, namely a name, its data type and the connection level for its aggregation. At the end of a CreateMultilevelFact statement, values and aggregation functions (e.g. SUM or AVG) for each measure can be defined. If an aggregation function is not specified by the user the data warehouse uses the summation function as default.

```
CREATE MULTILEVEL FACT BETWEEN OBJECTS (Product IN DIMENSION product_dim,
                                        Time IN DIMENSION time_dim,
                                           Location IN DIMENSION location_dim)
IN CUBE sales_cube
HIERARCHY (
   CONNECTION LEVEL <model, month, city> (
      revenue NUMBER
   )
)
SET revenue.function = SUM;
```

Listing 4.7: Create Multilevel Fact statement with adding a measure

```
CREATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter1, January2010, Salzburg)
IN CUBE sales_cube
SET revenue = 1000;
```

Listing 4.8: Create Multilevel Fact statement including set measure values

Listing 4.7 presents a CreateMultilevelFact statement that introduces the root m-fact for a sales cube, recognizable as the m-fact is situated at the m-cubes root coordinate. The statement further introduces a measure revenue for this m-fact. The connection level defines from where the measure aggregates its data. In this case all m-facts with a cube coordinate situated at <model, month, city> will be used. Such a m-fact is shown by the statement in Listing 4.8. The inserted m-fact sets a value for the measure revenue. Also an unqualified m-fact cube coordinate is used.

Alter multilevel object

With the help of AlterMultilevelObject statements attributes can be added and dropped. At first a dimension and a m-object must be specified. After that, attributes can be added and dropped for this m-object. When adding an attribute an abstraction level has to be defined. The abstraction level defines where the attribute will be located. Also a data type is needed. For dropping an attribute only the attribute name is needed. Listing 4.9 shows an AlterMultilevelObject statement adding the attribute *price* at the level *model* with the data type NUMBER, and dropping the attribute *author*.

```
ALTER MULTILEVEL OBJECT Product IN DIMENSION product_dim
ADD ATTRIBUTE price NUMBER AT LEVEL model,
DROP ATTRIBUTE author;
```

Listing 4.9: Alter Multilevel Object statement

Alter multilevel fact

Using the AlterMultilevelFact statement measures can be added and dropped. First the m-fact, where the measure shall be inserted, has to be specified. Therefore the m-fact cube coordinate is needed. It can be stated qualified or unqualified, just as explained at the CreateMultilevelFact statement. Also the name of the m-cube is needed where the m-fact is located in. After this, arbitrary measures can be added and dropped for this m-fact. For adding a measure, a name has to be specified for it and a data type. At last the connection level must be defined for the measure. Listing 4.10 shows a AlterMultilevelFact statement adding the measure *numberOfSoldItems* and dropping the measure *revenue*.

```
ALTER MULTILEVEL FACT BETWEEN OBJECTS (Product IN DIMENSION product_dim,
                                       Time IN DIMENSION time_dim,
                                       Location IN DIMENSION location_dim)
IN CUBE sales_cube
ADD MEASURE numberOfSoldItems NUMBER AT CONNECTION LEVEL <model, month, city>,
DROP MEASURE revenue;
```

Listing 4.10: Alter Multilevel Fact statement

Drop dimension hierarchy

By using the DropDimensionHierarchy one can delete a dimension hierarchy. The statement only consists of the drop clause and the dimension name which should be deleted. Listing 4.11 shows an example of a DropDimensionHierarchy statement.

```
1 DROP DIMENSION HIERARCHY product_dim;
```

Listing 4.11: Drop Dimension Hierarchy statement

Drop multilevel object

With the help of the DropMultilevelObject statement no longer required m-objects can be deleted. Therefor at first the m-object name has to be specified after the drop clause. Then the dimension hierarchy name where the m-object is situated must be stated. Listing 4.12 shows an example of a DropMultilevelObject statement.

```
1 DROP MULTILEVEL OBJECT Book IN DIMENSION product_dim;
```

Listing 4.12: Drop Multilevel Object statement

Drop multilevel cube

The DropMultilevelCube statement is used for deleting m-cubes. The statement only consists of the drop clause and the name of the m-cube which should be deleted. Listing 4.13 shows an example of a DropMultilevelCube statement.

```
1 DROP MULTILEVEL CUBE sales_cube;
```

Listing 4.13: Drop Multilevel Cube statement

Drop multilevel fact

For deleting a m-fact the DropMultilevelFact statement is used. It starts with the drop clause followed by the m-facts cube coordinate. Afterwards the corresponding m-cube name of the cube coordinate must be specified. Listing 4.14 shows an example of a DropMultilevelFact statement.

```
1 DROP MULTILEVEL FACT BETWEEN OBJECTS (Product IN DIMENSION product_dim,
2                                       Time IN DIMENSION time_dim,
3                                       Location IN DIMENSION location_dim)
4 IN CUBE sales_cube;
```

Listing 4.14: Drop Multilevel Fact statement

4.2. Data manipulation

Update statements provide the functionality to set values of attributes and measures. For each a separate UPDATE statement exists, which are a now presented.

Update multilevel object

With the UpdateMultilevelObject statement values of attributes can be set, or changed if an entry was already made earlier. First the regarding m-object has to be defined and the dimension where it is located. After that the statement can contain arbitrary SET commands for attributes of the m-object. Listing 4.15 shows an UpdateMultilevelFact statement with two SET commands.

```
1 UPDATE MULTILEVEL OBJECT HarryPotter1 IN DIMENSION product_dim
2 SET price = 25,
3     auhor = 'JKRowling';
```

Listing 4.15: Update multilevel object statement

Update multilevel fact

To set the values of measures the UpdateMultilevelFact statement is used. At first the m-fact is defined that the measure belongs to. This is done by defining the cube coordinate of the m-fact, which can be stated qualified or unqualified (see CreateMultilevelFact). After that the name of the corresponding m-cube has to be defined. When this is done, arbitrary SET commands for every measure located at the m-fact can be assigned. Listing 4.16 displays an UpdateMultilevelFact statement setting a value for the measure *revenue*.

```
1 UPDATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter1, January2010, Salzburg)
2 IN CUBE sales_cube
3 SET revenue = 1200;
```

Listing 4.16: Update multilevel fact statement

4.3. Data querying

With the SQL($\mathcal{M}$) query functionality the data inside the hetero-homogeneous data warehouse can be analysed. The SQL($\mathcal{M}$) query language is embedded into the SQL SELECT statement. The ROLLUP operator in SQL($\mathcal{M}$) bridges the gap between the hetero-homogeneous data warehouse and traditional relational OLAP. The result of the ROLLUP operator is a relational table. The ROLLUP operator can be used in the SQL FROM clause and takes the following input:

1. The aggregation level of the result specified as cube coordinate.
2. The multilevel cube that serve as the basis for aggregation.

Listing 4.17 shows a simple SQL query statement, which uses the ROLLUP operator from SQL($\mathcal{M}$) . It consists of a SELECT part and a FROM part. In the SELECT part the result table can be projected as in ordinary SQL queries. Note that a WHERE clause could also be used to further filter the result. The result table contains a column for each dimension of the m-cube and a column for each measure of the m-cube. In our example the whole result table will be displayed, as no selection criteria is chosen; the asterisk select all columns from the table. Of course it would also be possible to manually select all the wanted columns. The ROLLUP operator in the example aggregates the data from the *sales_cube* to the *model* } *month* } *city* aggregation level.

```
1 SELECT *
2 FROM ROLLUP[product_dim.model, time_dim.month, location_dim.city](sales_cube);
```

Listing 4.17: SQL($\mathcal{M}$) query to retrieve whole cube data

The following query examples work on the example data as shown in Figure 4.1. Furthermore, m-facts and corresponding measure values are set for all possible m-object combinations. The table at the bottom of Figure 4.1 displays the base data. Table 4.1 shows how the result of the query from Listing 4.17 would look like. Since the query aggregation level of the ROLLUP operator of the query example corespondes to the aggregation level of the base data, the result table is the same as the table in Figure 4.1; no actual aggregation is performed.

product_dim	**time_dim**	**location_dim**	**revenue**
HarryPotter1	January2010	Salzburg	1000
HarryPotter1	January2010	Zurich	2000
HarryPotter2	January2010	Salzburg	1500
HarryPotter2	January2010	Zurich	2500
Audia A3	January2010	Salzburg	1200000
Audi A3	January2010	Zurich	2000000

Table 4.1.: Query result of Listing 4.17

The query in Listing 4.18 illustrates how to change the granularity of a query result. In this query the abstraction level of dimension *product_dim* is set to level *category*.

Therefore the revenue is not displayed for each individual book and car, but only for their corresponding categories as a whole. This is accomplished with the ROLLUP operator. It allows the user to define the desired abstraction level for each dimension of the result. It is specified right after the ROLLUP key word between square brackets. The order of the levels can be defined qualified, by stating the corresponding dimension for each level. In this case the order of the levels irrelevant. When stating the levels unqualified, the order of dimensions has to be the same as at the m-cube's cube coordinate, defined at the creation of the m-cube. In this case first the level of the *product_dim*, the *time_dim* and at last the *location_dim* dimension is stated.

```
1 SELECT product_dim, time_dim, location_dim, revenue
2 FROM ROLLUP[product_dim.category, time_dim.month,
    location_dim.city](sales_cube);
```

Listing 4.18: SQL($\mathcal{M}$) query using more abstract levels at the ROLLUP

Table 4.2 shows the query result of Listing 4.18. When compared with Figure 4.1, the impact of the different abstraction levels of the two queries becomes clear. The values of the measure *revenue* are summed up from the individual values of the corresponding m-facts from the measure's connection level. For the calculation of measure values on higher abstraction levels different functions are available like sum or average. Which function is applied is defined at the creation of a measure.

product_dim	**time_dim**	**Location_dim**	**revenue**
Book	January2010	Salzburg	2500
Car	January2010	Salzburg	1200000
Book	January2010	Zurich	4500
Car	January2010	Zurich	2000000

Table 4.2.: Query result of Listing 4.18

Listing 4.19 shows how closed m-cube queries may be used to narrow the considered base data that serves as input for the ROLLUP operator. Therefore, at the place where in Listing 4.18 the m-cube's name was defined, now a closed m-cube query is stated. A closed m-cube query consists of three parts, namely SELECT, FROM and WHERE clause. A closed m-cube query may only be used as input for the ROLLUP operator or as a subquery in the FROM clause of a closed m-cube query. The key word sequence 'SELECT MULTILEVEL' indicates the beginning of a closed m-cube query. In the SELECT clause of the closed m-cube query the desired measures can be defined explicitly or one can use the asterisk to select all measures from the input m-cube. This corresponds to the projection operator from the closed m-cube query algebra [NST10]. The FROM clause selects an input m-cube which can be stated as the name of an m-cube or it can be given as a closed m-cube subquery. In the FROM clause the DICE operator allows for the selection of a subcube which also has a corresponding operator in the closed m-cube query algebra [NST10]. The DICE operator takes as input an cube coordinate. The cube coordinate can be stated qualified by defining the desired m-objects with the corresponding dimension. In this case the order of the m-objects is irrelevant. When

stating only the m-object names, the order of the m-objects need to be the same as defined by the cube's coordinate. Only the defined m-objects and their child m-objects at any lower abstraction level are considered for the result. The WHERE clause of a closed m-cube query corresponds to the slice operator of the closed m-cube query in the closed m-cube query algebra [NST10]. The WHERE clause is optional. It allows for the selection of specific cells of the cube which satisfy certain conditions. These conditions are expressed over the attributes of the coordinate m-objects.

```
1 SELECT *
2 FROM ROLLUP[product_dim.category, time_dim.month, location_dim.city]
3                         (SELECT MULTILEVEL revenue
4                          FROM  DICE[Book, January2010, Salzburg] (sales_cube) s
5                          WHERE s.product_dim.model.price < 50);
```

Listing 4.19: SQL($\mathcal{M}$) query with slice and dice expressions

In the query shown in Listing 4.19, the ROLLUP operation sets the m-cube on nearly the lowest abstraction level possible. In this example the ROLLUP operator rolls up the base data only along the *product_dim* dimension namely to *category* level. As input of the ROLLUP operator a subquery is specified. The SELECT clause of the subquery selects only the measure *revenue*. Other existing measures will not be included in the result. In the FROM clause the DICE operator narrows down the considered base data by defining a dice coordinate, i.e. a m-object for each dimension. The input of the subquery is the *sales_cube* m-cube. The WHERE clause specifies a constraint over the attribute *price* at the level model. Thus, the only considered m-objects at the level *model* are those with an attribute value of the attribute *price* smaller than 50.

product_dim	**time_dim**	**location_dim**	**revenue**
Book	January2010	Salzburg	2500

Table 4.3.: Query result of Listing 4.19

Looking at Table 4.3, one can see the effects of the dice, slice and projection operations of the query shown in Listing 4.19. Through the DICE operator the dimension *product_dim* is set to the *category* level. Thus the result does not contain individual products (e.g. HarryPotter1 or AudiA3) but *category*-level m-objects (e.g. Book and Car). The SELECT clause of the sub cube defines the measure *revenue* to be included in the result. The values of the measure revenue are summed up as defined by the dice operation. Then the slice operation refines the result again, by setting a constraint for the the values of the attribute *price* at level *model*. As a result the m-facts containing the m-object *Car* are disregarded, as no car with a price lower then 50 is existing in the example.

For the ROLLUP operator in SQL($\mathcal{M}$) queries also conversion functionality is available. The conversion of measure values is defined by using the CONVERT clause. The CONVERT clause is appended at the end of the ROLLUP specification. To be able to convert measure values conversion rules must be defined. This is done by defining separate m-cubes with measures representing the conversion mapping. This cubes are refered to as conversion cubes. A conversion cube always consists of a quantity dimension, which

hierarchically organizes units of measurement. Optionally, the conversion cube may also have a subset of the dimensions of the m-cube of interest the measure values of which should be converted. These additional dimensions allow for the definition of time-variant, location-variant, etc., conversion rates. The cube of interest and the conversion cube are then joined over the common dimensions. Before defining the conversion mapping the underlying dimensions and m-objects are needed. A good example for this concept are currencies. Listing 4.20 illustrates the dimension *quantity_dim*. The dimension consists of the level hierarchy *top* } *quantity* } *unit* with the m-object *Quantity* at the level *top*, *Currency* at level *quantity* as well as *Euro* and *Dollar* at level *unit*. The structure of the dimension *quantity_dim* is flexible and could be extended with other quantities like measures of length such as meter and feet for defining a conversion between the metric system and English units of measure.

```
1  CREATE DIMENSION HIERARCHY quantity_dim;
2
3  CREATE MULTILEVEL OBJECT Quantity IN DIMENSION quantity_dim AT LEVEL top
4  HIERARCHY (
5     LEVEL top CHILD OF NULL,
6     LEVEL quantity CHILD OF top,
7     LEVEL unit CHILD OF quantity
8  );
9
10 CREATE MULTILEVEL OBJECT Currency IN DIMENSION quantity_dim AT LEVEL quantity
       UNDER Quantity
11 HIERARCHY (
12    LEVEL quantity CHILD OF top,
13    LEVEL unit CHILD OF quantity
14 );
15
16 CREATE MULTILEVEL OBJECT Euro IN DIMENSION quantity_dim AT LEVEL unit UNDER
       Currency
17 HIERARCHY (
18    LEVEL unit CHILD OF quantity
19 );
20
21 CREATE MULTILEVEL OBJECT Dollar IN DIMENSION quantity_dim AT LEVEL unit UNDER
       Currency
22 HIERARCHY (
23    LEVEL unit CHILD OF quantity
24 );
```

Listing 4.20: dimension quantity_dim and corresponding m-objects

The dimension *quantity_dim* can be used to define a multilevel cube with measures defining the time-variant exchange rates between Euro and Dollar. Listing 4.21 shows the statements for creating such a *currency_cube*, which serves as conversion cube in the running example. The cube consists of the two dimensions *time_dim* and *quantity_dim*. For each *unit*-level m-object in the quantity dimension, in this example *quantity_dim*, a measure is defined. In this example for each currency a measure is created. The m-facts define for every currency/time pair the corresponding exchange rates.

```
CREATE MULTILEVEL CUBE currency_cube OF DIMENSIONS
   Currency IN DIMENSION quantity_dim,
   Time IN DIMENSION time_dim;

CREATE MULTILEVEL FACT BETWEEN OBJECTS (Currency, Time)
IN CUBE currency_cube
HIERARCHY (
   CONNECTION LEVEL <unit, month> (
      Euro NUMBER,
      Dollar NUMBER
   )
)
SET  Euro.function = AVG DEFAULT,
   Dollar.function = AVG DEFAULT;

CREATE MULTILEVEL FACT BETWEEN OBJECTS (Euro, January2010) IN CUBE
    currency_cube
SET  Euro = 1, Dollar = 1.40;

CREATE MULTILEVEL FACT BETWEEN OBJECTS (Dollar, January2010) IN CUBE
    currency_cube
SET  Euro = 0.6, Dollar = 1;
```

Listing 4.21: SQL($\mathcal{M}$) query with measure value conversion

To be able to use the defined conversion data inside a query, the to be converted measure must be defined as such. This is done by defining a unit for the respective measure. Listing 4.22 shows how to set the unit to enable unit conversion.

```
UPDATE MULTILEVEL FACT BETWEEN OBJECTS (Product, Time, Location) IN CUBE
    sales_cube
SET revenue.unit = Euro IN DIMENSION quantity_dim DEFAULT;
```

Listing 4.22: Setting a unit for measure conversion

Listing 4.23 displays a SQL($\mathcal{M}$) query containing the CONVERT functionality. The query starts with the mandatory SELECT clause. The ROLLUP expression defines the level of abstraction of the query result. After that a ClosedMCubeQuery is defined, where the selected measures are restricted to the measure *revenue*. The following DICE operator further narrows down the considered base data to the m-facts under the given dice coordinate. At the end of the query the conversion functionality recalculates all measure values inside the result to Dollar values as defined by the measure *Dollar* in the m-cube *currency_cube*.

```
SELECT *
FROM ROLLUP[product_dim.model, time_dim.month, location_dim.city](
SELECT MULTILEVEL revenue
FROM DICE[Book, January2010, Salzburg] (sales_cube)
)
CONVERT MEASURE revenue TO Dollar IN quantity_dim USING currency_cube;
```

Listing 4.23: SQL($\mathcal{M}$) query with measure value conversion

The query returns values of the measure *revenue* for books sold in January 2010 in the city of Salzburg. As base data serve the example data in Figure 4.1. Table 4.4 shows the result of the query without applying the conversion. Table 4.5 displays the result of the query when the measure values are converted from Euro to Dollar.

product_dim	**time_dim**	**location_dim**	**revenue**
HarryPotter1	January2010	Salzburg	1000
HarryPotter2	January2010	Salzburg	1500

Table 4.4.: Query result of Listing 4.23 without conversion

product_dim	**time_dim**	**location_dim**	**revenue**
HarryPotter1	January2010	Salzburg	1400
HarryPotter2	January2010	Salzburg	2100

Table 4.5.: Query result of Listing 4.23

The conversion from Euro to Dollar for the month *January2010* is defined in the m-cube *currency_cube* (see Listing 4.21). The value of the measure *Dollar* is set to 1.4 at the m-fact Euro, January2010, which is applied during the conversion. Therefore all measure values returned by the query are multiplied by 1.4 for the final result.

5. An interpreter for SQL($\mathcal{M}$)

After an introduction of the theoretical basics and the clarification of the motivation behind the project, the implementation of the interpreter will now be presented in detail. At the beginning the focus lies on the design of the interpreter. After that the practical realization will be explained in detail.

5.1. Approach

As already mentioned, a main problem of the hetero-homogeneous data warehouse was the cumbersome interaction possibilities for end users. The solution of the problem was the development of an easy to use interaction language. For faster an intuitive handling, the language is based on SQL and is extended with the needed multilevel OLAP functionalities and is therefore called SQL-Multilevel (SQL($\mathcal{M}$)). The language extansion contains statements for declaring and updating the multilevel structure of the hetero-homogeneous data warehouse and the querying of the data inside it. The language details are precisely defined in Chapter 4 and Appendix A. Based on the SQL($\mathcal{M}$) language the SQL($\mathcal{M}$) interpreter is conceived, for taking the SQL($\mathcal{M}$) input and transforming it into PL/SQL code the hetero-homogeneous data warehouse prototype can process.

This chapter now explains the technological decisions leading to the SQL($\mathcal{M}$) interpreter and why certain concepts were preferred over others. These include decisions like choosing a compiler and decide which programming language to use, which affect the overall system design and interpreter design. For making these decisions the following criteria were crucial.

First, a simple compiler is not sufficiant, but a interpreter for executing the translated code is needed. The hetero-homogeneous data warehouse prototype dynamicially ceates object types and PL/SQL code. A compiler, before translating one SQL($\mathcal{M}$) statement into PL/SQL code, must take into account the effects of the previous SQL($\mathcal{M}$) statements. Thus, a compiler could only translate one SQL($\mathcal{M}$) statement at a time. The proposed interpreter takes a batch of SQL($\mathcal{M}$) statements, then translates into PL/SQL code one statement at a time. Before the translation of one statement the previous statement is executed, which allows the interpreter to take into account possible changes to the database schema.

Second, the interpreter should be able to perform optimizations on the SQL($\mathcal{M}$) input to generate faster PL/SQL code, BULK statements provided by the hetero-homogeneous data warehouse prototype. BULK statements merge together individual statements which are compatible with each other and provide a better performance than the individual execution of the statements. This optimization must be performed between the parsing of the input and its actual translation. Therefore a local representation of the SQL($\mathcal{M}$) input stream is needed to be able to search the input for optimization options and adapt it when needed.

Third, the hetero-homogenous data warehouse prototype is implemented by extending an Oracle database using its object-relational features. This means that many Oracle

features, such as the Java Virtual Machine (JVM) provided by the database, should be considered by the selection of a solution.

The interpreter is realized as a Java stored procedure. Thereby, the whole functionality needed can be realized in one package running directly inside the database. This solution has the following benefits:

- The interpreter must not be handled separately by the user. Once loaded into the database, the SQL($\mathcal{M}$) interpreter is centraly available to many client applications.
- A Java stored procedure can use a special server-side internal driver for communicating with the oracle database. It uses the user's default session and provides an uncomplicated access to the database. Further it is optimized to run within the database server and thereby enhances the performance. [Ora09, p. 1-17]

The programming the languages that can be used for realising the interpreter s limited to Java. Therefore, a Java parser generator was selected for the implementation of the compiler functionalities. An important factor for this decision was the design of the SQL($\mathcal{M}$) language. SQL($\mathcal{M}$) is mainly defined as a LL(1) grammar, but it contains also some LL(k) constructs. Therefore a LL(k) Java parser generator was needed to represent the entire SQL($\mathcal{M}$) language grammar. Three popular tools were examined more closely, namely the Java Compiler Compiler (JavaCC), COCO/R (Lexer and Parser Generators) and ANTLR (Another Tool for Language Recognition).

The first important difference between the considered tools appeared when comparing the provided functionality for defining a grammar. Here JavaCC offers the possibility of a local lookahead. Using a local lookahead grammar parts with a higher lookahead can be integrated into LL(1) grammars without being forced to declare the whole definition as LL(k). This option has an impact on the performance of the lexical analysis. Looking at the SQL($\mathcal{M}$) grammar, as already mentioned, most of the grammar parts are distinguishable with a lookahead of one and only a small portion of the grammar needs a lookahead of two grammar expressions. Without a local lookahead for the specific LL(2) grammar parts the whole grammar must be declared as an LL(2) grammar. Therefore all grammar rulings would be decided by looking two tokens ahead although most of the times one token would be sufficient. That implies that without the local lookahead the lexical analysis would perform poorer.

At the same time the question of how to design the structure of the interpreter and the whole translation process was important and influenced the parser generator decision. For the translation process, several different approaches were possible. First, a direct translation was considered. In this case the application would parse and translate the input in one go. This variant was dropped because of the specification that the interpreter should support the optimization functions of the data warehouse prototype. Therefore an analysis and modification of the SQL($\mathcal{M}$) input is necessary which cannot be achieved by a direct translation. So a translation based on an internal representation was examined next. Here JavaCC and ANTLR revealed an advantage compared to COCO/R, as they support the automatic generation of syntax trees, representing the processed data input. JavaCC and ANTLR create different variants of syntax trees. JavaCC allows for the automatic generation of an abstract syntax tree (AST) whereas ANTLR allows for the automatic generation of a parse tree. Quelle For a more detailed discussion on the differences of AST and parse tree [Ter96, p. 23 et seq.]. Both could be used for the realization of the SQL($\mathcal{M}$) interpreter.

The decision was made in favor of the Java Compiler Compiler (JavaCC), as it covers all the functionality needed for the implementation. For example, it provides a local lookahead for the parsing process and auto generated abstract syntax trees. Also, JavaCC is widely in use and a large amount of examples and documentation is available.

After the decision to use JavaCC as the compiler implementation framework, the next step was to design the whole translation process of the abstract syntax tree. The syntax tree represents the identified SQL($\mathcal{M}$) statements by using nodes. Each node stands for an identified grammar part of the SQL($\mathcal{M}$) input and contains the information of it. Figure 5.1 shows the layout of a syntax tree. Every statement is a sub-tree under the root node. These sub-trees may again contain several sub-trees which include the corresponding nodes that constitute the statements. Every input statement is deconstructed this way to its basic grammar parts during the lexical analysis.

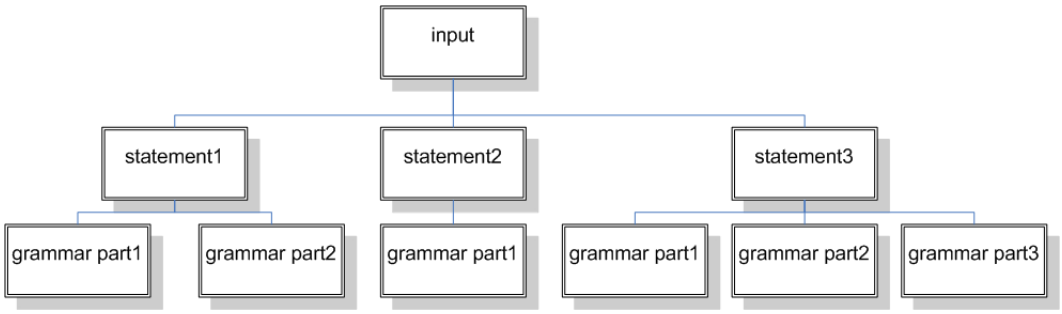


Figure 5.1.: Layout of an abstract syntax tree

For each type of statement (CREATE, ALTER, DROP, etc.) there exists a specific translation process. Each process is realized by a separate class which traverses the AST and conduct the actual translation. The visitor pattern could also be used and should be considered for future work.

5.2. System design

The SQL($\mathcal{M}$) interpreter is part of the hetero-homogeneous data warehouse system which consists of following three parts:

- hetero-homogenous data warehouse
- SQL($\mathcal{M}$) interpreter
- end-user

In this section the general system design is presented in detail by showing how the different pieces are working together.

The hetero-homogenous data warehouse is the basis of the system. It is realized by use of custom object types implemented in PL/SQL. Therefore all interaction with the data warehouse has to occur via PL/SQL. A user or an application can operate the data warehouse by using the functions provided by the custom data types. As already mentioned in Section 5.1 the usage via PL/SQL is kind of cumbersome for end-users. Therefore the SQL($\mathcal{M}$) interpreter is implemented to provide a user friendly interaction with the data warehouse. The SQL($\mathcal{M}$) interpreter is implemented as a Java application running

inside the Java virtual machine (JVM) of the Oracle database (DB). It takes the input of the end-user, translates it to the corresponding PL/SQL code and sends it to the data warehouse. Eventual query results or error messages are returned to the end-user.

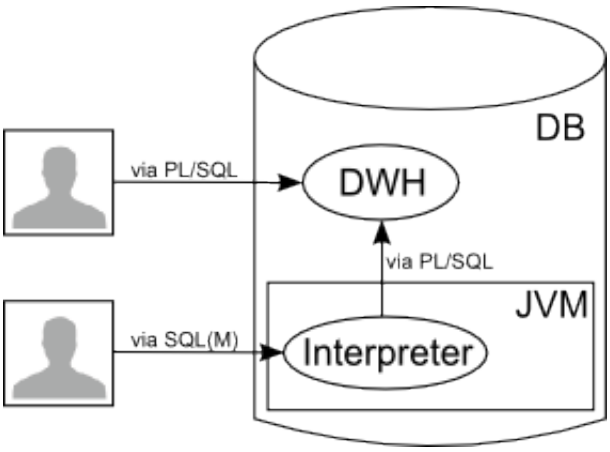


Figure 5.2.: System design

Figure 5.2 displays the positions of the different parts of the whole system and the interaction between them.

For users to be able to call the SQL($\mathcal{M}$) interpreter an application programming interface (API) is provided. The interpreter interface at the database level is realized as a Java stored procedure. As input, data of type CLOB (character large object) is expected. As result the interface returns a SQL ResultSet. For returning a ResultSet an Oracle object of type Ref Cursor is needed. Therefore at first a custom package with an object of type Ref Cursor is created. This is then used for defining the SQL function *sqlm_query* calling the counter part of the interface at the interpreter level.

```
CREATE OR REPLACE PACKAGE refcurpkg AS
  type refcur_ty is ref cursor;
end;

CREATE OR REPLACE FUNCTION sqlm_query(input CLOB) RETURN
   refcurpkg.refcur_ty IS
  LANGUAGE JAVA NAME
     'at.jku.dke.sqlm.interpreter.Interpreter.sqlmQuery(oracle.sql.CLOB)
     return java.sql.ResultSet';
```

Listing 5.1: Interpreter API definition at database level

When calling the function no further invoking has to be done for being able to use the SQL($\mathcal{M}$) interpreter. By establishing a database connection a corresponding database session is created. Each database session includes a JVM session. All Java applications located inside the database are thereby accessible [Ora09, 2-1 et seq.]. A brief instruction, of how to load a Java application into an Oracle database, is given at Appendix C.

Listing 5.2 gives an example of how the function *sqlm_query* can be used in a Java

application. The code is from the SQL($\mathcal{M}$) Editor (see Appendix C). The SQL($\mathcal{M}$) Editor is a JavaFX application for sending SQL($\mathcal{M}$) statements and displaying their results. Listing 5.3 displays how to use the function *sqlm_query* in PL/SQL.

```
String input = 'CREATE DIMENSION HIERARCHY Product_dim;';

Connection cn =
   DriverManager.getConnection("jdbc:oracle:thin:@IPADDRESS:PORT:DBINSTANCE",
                                          "USER",
                                          "PASSWORD");
CallableStatement stmt = cn.prepareCall("{? = call SQLM_QUERY(?)}",
                                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                                    ResultSet.CONCUR_UPDATABLE);

stmt.registerOutParameter(1, OracleTypes.CURSOR);
stmt.setString(2, input);
stmt.execute();
```

Listing 5.2: Using the interpreter API in Java applications

```
VARIABLE cur REFCURSOR
EXEC :cur := sqlm_query('SELECT * FROM ROLLUP[product_dim.category,
   time_dim.year, location_dim.country](sales_cube);');
print :cur
```

Listing 5.3: Using the interpreter API in Oracle

5.3. Interpreter design

Now the SQL($\mathcal{M}$) interpreter design will be brought into focus to give an overview over the application. Afterwards the structure of the interpreter implementation is shown and how the described process is covered by the SQL($\mathcal{M}$) interpreter. Before the design of an application can be discussed, the process it should cover needs to be known. Therefore the overall process of SQL($\mathcal{M}$) interpreter is presented first. The whole process consists of several sub-processes. The main sub-processes can be described as:

1. Reception and validation of an input stream.
2. Creation of an abstract syntax tree which represents the validated input.
3. Execution of optimizations inside the abstract syntax tree.
4. Compiling and execution of the statements provided by the abstract syntax tree.

The process is traditionally divided into two main parts: the *analysis* and the *synthesis*. The analysis is responsible for reading and verifying the input stream. The validation is done regarding the SQL($\mathcal{M}$) grammar and is called parsing. After the validation, the second task is the depiction of the identified statements in form of an abstract syntax

tree (AST). The AST is built-up by different node types, implemented as so called AST-classes. Each class represents a SQL($\mathcal{M}$) grammar part and holds the specific information of the validated data input. For supporting the development of the parsing process the Java Compiler Compiler (JavaCC) framework was chosen.

After the analysis the synthesis process starts. At first the optimization sub-process scans the AST for statements which can be merged. Thereby the translation process and, more important, the execution of the translated code inside the data warehouse becomes faster then the original one. After the AST is brought to its final shape, certain semantic checks are done. The implemented tests thereby only require data of the AST, no information at all of the data warehouse. This is important because additional data retrieval would slow down the process. Further required semantic tests are done by the data warehouse by default. For that reason these tests are ignored by the interpreter at this implementation level. Usually the task ends here for the synthesis phase, but in case of the interpreter, the synthesis is extended with the execution of the generated code.

As explained in detail in Section 5.1, the goal of the implementation was to create an interpreter, implemented in the programming language Java. Thereby it can run inside the Java virtual machine (JVM) of an Oracle enterprise edition database 11g R1 and can be accessed as an Java stored procedure. As a consequence the application code is written in Java version 1.5, as this is the highest supported Java version of the Oracle database 11g R1 [jav, p. 5]. The JavaCC framework is used in version 5.0, which was the up-to-date version at implementation start and is compatible with Java version 1.5.

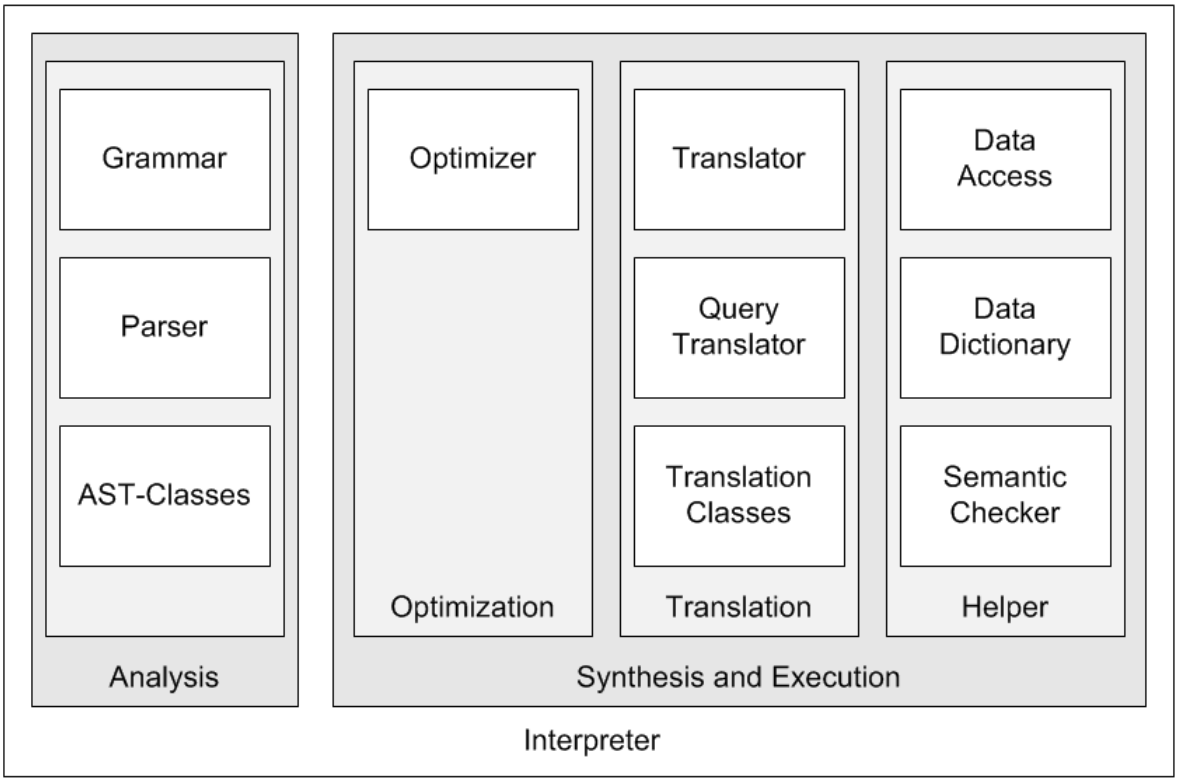


Figure 5.3.: interpreter architecture design

The interpreter is subdivided into parts shown in Figure 5.3. The different sub-processes of the analysis and synthesis parts are represented through the functionalities of the different classes provided by the interpreter. The classes are clustered among themselves

to packages. Each package provides the functionality for a certain sub-process. The implementation of the packages and their functionality is presented in detail in the next section.

5.4. Interpreter implementation

After the presentation of the interpreter design now the implementation of the different interpreter parts will be illustrated in detail. The individual parts will be presented in the same sequence as used during the overall process. Hence the *analysis* is presented first, followed by the *synthesis* and *execution*. At the end the API of the application is introduced.

5.4.1. Analysis

To be able to parse an input the parser needs a grammar definition. A grammar definition allows the parser to disassemble an input under fixed rules and evaluate its correctness (see Section 2.1). Before a grammar can be defined a language is needed.

So the first step at the parser development was the definition of the SQL($\mathcal{M}$) query language. As a starting point served the functionality of the already existing data warehouse, which predetermined to scope of the SQL($\mathcal{M}$) query language. As reference point for the language design, ordinary SQL was chosen. The intention was to facilitate the usability of the SQL($\mathcal{M}$) language, as the command and clause names (CREATE, UPDATE, WHERE, and so on) of SQL are commonly known. The language is presented in detail in Chapter 4.

The implementation of the SQL($\mathcal{M}$) grammar is done with the JavaCC framework. In JavaCC the definition of a grammar is done inside a jjt-file. Inside this file all needed definitions for the SQL($\mathcal{M}$) grammar are implemented. The file thereby contains following definitions:

- terminal symbols
- non terminal symbols
- start symbol
- production rules

The SQL($\mathcal{M}$) grammar is defined as a context-free grammar (see Section 2.1). Contextfree grammars define production rules the following way. At the left side of a production rule only one non terminal symbol is allowed. At the right side arbitrary numbers of terminal and non terminal symbols can be situated. The SQL($\mathcal{M}$) language is defined by nesting production rule within production rule of the SQL($\mathcal{M}$) grammar.

The grammar parts for the data definition and data manipulation statements are implemented from scratch. the definition of the grammar for data query statements was more complex. Here the idea was to extend the SQL SELECT statement with the functionality needed for SQL($\mathcal{M}$) queries. Therefore already implemented SQL grammars were taken and integrated into the SQL($\mathcal{M}$) grammar definition. The scope of the integration is limited to the functionality needed for the SQL($\mathcal{M}$) language (see Appendix A). Inside the SQL($\mathcal{M}$) grammar definition the copied grammar parts are highlighted and their origins are documented. The integration of the SQL($\mathcal{M}$) query grammar into SQL

is done by adding an additional command called ROLLUP to the SQL grammar. The command is defined directly after the SQL FROM clause and is the starting point for the SQL($\mathcal{M}$) query grammar. As a result a SQL($\mathcal{M}$) multilevel-cube can be defined instead of a database table or sub-query. Listing 5.4 displays an example of a SQL($\mathcal{M}$) query, showing the combination of a SQL SELECT statement with the SQL($\mathcal{M}$) query extension to query the multilevel cube *sales_cube*.

```
SELECT * FROM ROLLUP[product_dim.category, time_dim.year,
   location_dim.country](sales_cube);
```

Listing 5.4: SQL($\mathcal{M}$) query

From the grammar file the JavaCC framework auto generates a LL(k) parser. The k indicates the number of grammar expressions the parser needs to look at, before it can decide if the input is valid and which production rule is applied. Therefore the number of k depends on the structure of the grammar definition. In case of the SQL($\mathcal{M}$) grammar a LL(1) parser with a local LL(2) is generated. The user defined local lookahead is one of the key features of the JavaCC framework. Due to the local lookahead not the whole parsing process has to run as LL(2), but only the necessary grammar parts. This has a positive impact on the performance as the decision making of parser is as efficient as possible.

The generated parser is used to validate an input stream and transform it to an abstract syntax tree (AST). For generating an AST the JavaCC framework provides the JJTree. JJTree can be seen as a preprocessor for generating an AST. JJTree creates a jj-file from the user-defined jjt-file. The JavaCC framework then processes the jj-file to build the AST. During the parsing process, for every invoked production rule of the SQL($\mathcal{M}$) grammar, an entry is created inside the AST. The entries are called nodes. The type of a node is derived from the name of the corresponding production rule. By default the prefix AST is put in front of every node type. All defined nodes must implement the JJTree Java interface *Node*. The interface specifies the functionality for all node types. Methods like setting parent nodes or adding and retrieving children nodes. With them the creation, selection and manipulation of the AST is given [jcc].

```
ASTSQLMDocument document = SQLMParser.Start();
```

Listing 5.5: Starting of the parsing process

Listing 5.5 displays how the parsing process is started in Java. To start the parsing an object of type ASTSQLMDocument is needed. The abstract syntax tree of a SQL($\mathcal{M}$) input always starts with a node of type ASTSQLMDocument as it is the so called root node and is the equivalent of the start symbol of the SQL($\mathcal{M}$) grammar. With *SQLMParser.Start()* the parsing process starts and builds up the AST under the root node. This way all identified SQL($\mathcal{M}$) statements are gathered during the parsing process under the ASTSQLMDocument node.

The following listings illustrate an example of a generated AST of the SQL($\mathcal{M}$) parser. Listing 5.6 displays an input example. The input consists of five SQL($\mathcal{M}$) statements. Listing 5.7 shows the generated AST for these five statements. Each entry of the AST is

either a terminal or non terminal symbol. For terminal symbols the corresponding value is displayed inside square brackets. The indents of the entries represent the production rules of the SQL($\mathcal{M}$) grammar. The entry on a higher level represents the left side of a production rule and the entries directly below the right side.

```
CREATE DIMENSION HIERARCHY product_dim;

CREATE MULTILEVEL OBJECT Product IN DIMENSION product_dim AT LEVEL top
HIERARCHY (
   LEVEL top CHILD OF NULL,
   LEVEL category CHILD OF top,
   LEVEL model CHILD OF category
);

CREATE MULTILEVEL OBJECT Book IN DIMENSION product_dim AT LEVEL category
    UNDER Product
HIERARCHY (
   LEVEL category CHILD OF top,
   LEVEL model CHILD OF category (
      author VARCHAR2(40)
   )
);

CREATE MULTILEVEL OBJECT DaVinciCode IN DIMENSION product_dim AT LEVEL
    model UNDER Book
SET author = 'Brown';

CREATE MULTILEVEL OBJECT HarryPotter IN DIMENSION product_dim AT LEVEL
    model UNDER Book;
SET author = 'Rowling';
```

Listing 5.6: Parsing: SQL($\mathcal{M}$) input

```
SQLMDocument
   CreateDimensionHierarchy
      DimensionHierarchyID [product\_dim]
    CreateMultilevelObject
       MultilevelObjectQualifiedID
          MultilevelObjectUnqualifiedID  [Product]
          DimensionHierarchyID [product\_dim]
  MultilevelObjectLevelID  [top]
  MultilevelObjectLevelHierarchy
       MultilevelObjectLevelDefinition
          MultilevelObjectLevelID  [top]
          MultilevelObjectLevelParentLevels
       MultilevelObjectLevelDefinition
          MultilevelObjectLevelID  [category]
          MultilevelObjectLevelParentLevels
             MultilevelObjectLevelID  [top]
    MultilevelObjectLevelDefinition
```

```
            MultilevelObjectLevelID [model]
            MultilevelObjectLevelParentLevels
                MultilevelObjectLevelID [category]
CreateMultilevelObject
    MultilevelObjectQualifiedID
        MultilevelObjectUnqualifiedID [Book]
        DimensionHierarchyID [product\_dim]
    MultilevelObjectLevelID [category]
    MultilevelObjectParents
        MultilevelObjectUnqualifiedID [Product]
    MultilevelObjectLevelHierarchy
        MultilevelObjectLevelDefinition
            MultilevelObjectLevelID [category]
            MultilevelObjectLevelParentLevels
                MultilevelObjectLevelID [top]
    MultilevelObjectLevelDefinition
        MultilevelObjectLevelID [model]
        MultilevelObjectLevelParentLevels
            MultilevelObjectLevelID [category]
        MultilevelObjectLevelAttribute
            MultilevelObjectAttributeID [author]
            Varchar2Type [VARCHAR2]
                DataLength [40]
CreateMultilevelObject
    MultilevelObjectQualifiedID
        MultilevelObjectUnqualifiedID DaVinciCode
        DimensionHierarchyID [product\_dim]
    MultilevelObjectLevelID [model]
    MultilevelObjectParents
        MultilevelObjectUnqualifiedID [Book]
    MultilevelObjectAttributeValueBlock
        MultilevelObjectAttributeValueAssignment
            MultilevelObjectAttributeID [author]
            StringValue ['Brown']
CreateMultilevelObject
    MultilevelObjectQualifiedID
        MultilevelObjectUnqualifiedID [HarryPotter]
        DimensionHierarchyID [product\_dim]
    MultilevelObjectLevelID [model]
    MultilevelObjectParents
        MultilevelObjectUnqualifiedID [Book]
    MultilevelObjectAttributeValueBlock
        MultilevelObjectAttributeValueAssignment
            MultilevelObjectAttributeID [author]
            StringValue ['Rowling']
```

Listing 5.7: Parsing: generated AST

5.4.2. Synthesis and execution

After the analysis of the input, the synthesis process starts. The purpose of the synthesis process is to transform SQL($\mathcal{M}$) statements into PL/SQL code processible by the hetero-homogeneous data warehouse. The following sections describe the synthesis process in chronological order and on the basis of the developed Java classes. Therefore first the optimization of the abstract syntax tree is presented. After that the actual translation of the SQL($\mathcal{M}$) statements is described. Auxiliary the implemented assisting functionalities for the translation are presented including the execution of the generated code.

Optimizer

For receiving better execution time an optimization in form of bulk statements is realized. The hetero-homogenous data warehouse provides the bulk statement functionality for four types of SQL($\mathcal{M}$) statements. For all four statement types the optimization process was developed. The functionality is implemented in the Java class *Optimizer*. The goal of the optimization process is to identify SQL($\mathcal{M}$) statements inside the abstract syntax tree which can be merged into bulk statements. When such statements are found, the optimization process rearranges the abstract syntax tree, so that it represents the identified bulk statements. The class *Optimizer* contains one private method for each statement type an optimization is possible. Those methods are called by the public method *run* executing them in a determined sequence.

The design of the different optimization processes is the same. They analyse the nodes found under the forwarded root node (ASTSQLMDocument) delivered through the *run(SQLMNode sn)* method. Each of the optimization methods thereby only looks after information for one explicit bulk optimization. When a potential statement is found, a key is generated from its data. The key is used to compare the statements of one type and decide if they can be merged together or not. Table 5.4.2 shows which statement types are analyzed by the different optimization processes. Additionally the composition of the validation keys is listed.

Optimization	Source statements	Key data
bulk create multi-level object	• CreateMultilevelObject statements	• dimension name • top-level • parent names • level hierarchy • grouping information
bulk update multi-level object	• SET attribute information inside of CreateMultilevelObject statements • UpdateMultilevelObject statements	• dimension name • attribute name • grouping information
bulk create multi-level fact	• CreateMultilevelFact statements	• cube name • multilevel fact hierarchy • grouping information
bulk update multi-level fact	• SET attribute information inside of CreateMultilevelFact statements • UpdateMultilevelObject statements	• cube name • measure name • grouping information

Table 5.1.: Analyzed SQL($\mathcal{M}$) statements per optimization process

The information a key consists of is the minimum set of information needed to decide if the data inside of two statements can be merged together to one bulk statement. Further the keys contain grouping information, which indicates if the statements related to the compared keys are next to each other inside the abstract syntax tree. This grouping is done fore keeping the semantic consistency, defined by the user, through the execution of arbitrary bulk statements. The generated keys are saved inside a list. Furthermore the positions of the statements corresponding to a key are saved. After all statements inside the abstract syntax tree are analysed for possible optimizations, for all keys with more than one relating statement a bulk statement is created. The bulk statement is inserted into the AST at the same position as the individual statements were positioned. The

individual statements are repositioned under the bulk statement. During the optimization process of update statements also create statements are taken into account as they can hold update information. Is update information found inside a create statement, then for this data a new container is created. This container is then filled with all the needed information from the create statement to get a self-sustained update statement. The part of the syntax tree with the update information is reallocated under the new container. The information inside the create statement is deleted, so that no duplicates remain. The so created update statement is then inserted under the bulk statement.

The following listings give an example of the impact the optimization process has on an AST. Listing 5.8 display an AST before the optimization and Listing 5.9 the outcome of the optimization process. As input the same SQL($\mathcal{M}$) statements are used as in the parsing example in Section 5.4.1.

```
SQLMDocument
    CreateDimensionHierarchy
        DimensionHierarchyID [product_dim]
    CreateMultilevelObject
        MultilevelObjectQualifiedID
            MultilevelObjectUnqualifiedID  [Product]
            DimensionHierarchyID [Product_dim]
   MultilevelObjectLevelID  [top]
   MultilevelObjectLevelHierarchy
        MultilevelObjectLevelDefinition
            MultilevelObjectLevelID  [top]
            MultilevelObjectLevelParentLevels
        MultilevelObjectLevelDefinition
            MultilevelObjectLevelID  [category]
            MultilevelObjectLevelParentLevels
                MultilevelObjectLevelID  [top]
     MultilevelObjectLevelDefinition
            MultilevelObjectLevelID  [model]
            MultilevelObjectLevelParentLevels
                MultilevelObjectLevelID  [category]
CreateMultilevelObject
    MultilevelObjectQualifiedID
        MultilevelObjectUnqualifiedID  [Book]
       DimensionHierarchyID [product_dim]
    MultilevelObjectLevelID  [category]
    MultilevelObjectParents
        MultilevelObjectUnqualifiedID  [Product]
    MultilevelObjectLevelHierarchy
        MultilevelObjectLevelDefinition
            MultilevelObjectLevelID  [category]
            MultilevelObjectLevelParentLevels
                MultilevelObjectLevelID  [top]
     MultilevelObjectLevelDefinition
        MultilevelObjectLevelID  [model]
        MultilevelObjectLevelParentLevels
            MultilevelObjectLevelID  [category]
        MultilevelObjectLevelAttribute
```

```
            MultilevelObjectAttributeID  [author]
          Varchar2Type [VARCHAR2]
             DataLength [40]
CreateMultilevelObject
   MultilevelObjectQualifiedID
      MultilevelObjectUnqualifiedID  DaVinciCode
     DimensionHierarchyID [product_dim]
   MultilevelObjectLevelID  [model]
   MultilevelObjectParents
      MultilevelObjectUnqualifiedID  [Book]
   MultilevelObjectAttributeValueBlock
      MultilevelObjectAttributeValueAssignment
         MultilevelObjectAttributeID  [author]
         StringValue  ['Brown']
CreateMultilevelObject
   MultilevelObjectQualifiedID
      MultilevelObjectUnqualifiedID  [HarryPotter]
     DimensionHierarchyID [product_dim]
   MultilevelObjectLevelID  [model]
   MultilevelObjectParents
      MultilevelObjectUnqualifiedID  [Book]
   MultilevelObjectAttributeValueBlock
      MultilevelObjectAttributeValueAssignment
         MultilevelObjectAttributeID  [author]
         StringValue  ['Rowling']
```

Listing 5.8: AST before optimization

```
SQLMDocument
   CreateDimensionHierarchy
      DimensionHierarchyID [product_dim]
   CreateMultilevelObject
      MultilevelObjectQualifiedID
         MultilevelObjectUnqualifiedID  [Product]
        DimensionHierarchyID [Product_dim]
      MultilevelObjectLevelID  [top]
      MultilevelObjectLevelHierarchy
         MultilevelObjectLevelDefinition
            MultilevelObjectLevelID  [top]
            MultilevelObjectLevelParentLevels
         MultilevelObjectLevelDefinition
            MultilevelObjectLevelID  [category]
            MultilevelObjectLevelParentLevels
               MultilevelObjectLevelID  [top]
         MultilevelObjectLevelDefinition
            MultilevelObjectLevelID  [model]
            MultilevelObjectLevelParentLevels
               MultilevelObjectLevelID  [category]
   CreateMultilevelObject
      MultilevelObjectQualifiedID
         MultilevelObjectUnqualifiedID  [Book]
```

```
        DimensionHierarchyID [product_dim]
    MultilevelObjectLevelID [category]
    MultilevelObjectParents
        MultilevelObjectUnqualifiedID [Product]
    MultilevelObjectLevelHierarchy
        MultilevelObjectLevelDefinition
            MultilevelObjectLevelID [category]
            MultilevelObjectLevelParentLevels
                MultilevelObjectLevelID [top]
        MultilevelObjectLevelDefinition
            MultilevelObjectLevelID [model]
            MultilevelObjectLevelParentLevels
                MultilevelObjectLevelID [category]
            MultilevelObjectLevelAttribute
                MultilevelObjectAttributeID [author]
                Varchar2Type [VARCHAR2]
                    DataLength [40]
BulkCreateMultilevelObject
    CreateMultilevelObject
        MultilevelObjectQualifiedID
            MultilevelObjectUnqualifiedID [DaVinciCode]
            DimensionHierarchyID [product_dim]
        MultilevelObjectLevelID [model]
        MultilevelObjectParents
            MultilevelObjectUnqualifiedID [Book]
    CreateMultilevelObject
        MultilevelObjectQualifiedID
            MultilevelObjectUnqualifiedID [HarryPotter]
            DimensionHierarchyID [product_dim]
        MultilevelObjectLevelID [model]
        MultilevelObjectParents
            MultilevelObjectUnqualifiedID [Book]
BulkUpdateMultilevelObject
    UpdateMultilevelObject
        MultilevelObjectQualifiedID
            MultilevelObjectUnqualifiedID [DaVinciCode]
            DimensionHierarchyID [product_dim]
        MultilevelObjectAttributeValueBlock
            MultilevelObjectAttributeValueAssignment
                MultilevelObjectAttributeID [author]
                StringValue ['Brown']
    UpdateMultilevelObject
        MultilevelObjectQualifiedID
            MultilevelObjectUnqualifiedID [HarryPotter]
            DimensionHierarchyID [Product_dim]
        MultilevelObjectAttributeValueBlock
            MultilevelObjectAttributeValueAssignment
                MultilevelObjectAttributeID [author]
                StringValue ['Rowling']
```

Listing 5.9: AST after optimization

The example above shows how the optimization process transforms individual statements into bulk statements. Affected are the last two *CreateMultilevelObject* statements in Listing 5.8. These two are transformed into a *BulkCreateMultilevelObject* and *BulkUpdateMultilevelObject* statement. The optimization process took the update information of both create statements and transformed it to the *BulkUpdateMultilevelObject* shown at the bottom of Listing 5.9. Then the update information was deleted from the create statements and the remaining data of the two create statements were combined to the *BulkCreateMultilevelObject*. Listing 5.10 shows the keys generated during the optimization process and illustrates why the information of these two statements were transformed into the bulk statements shown in Listing 5.9. On the basis of the generated keys the optimization process decided which statements to merge. Comparing the keys of the last two *CreateMultilevelObject* statements one can see that those match.

```
//create m-object Product//
product_dim/top/NULL/top-category-model/0
//create m-object Book//
product_dim/category/Product/category-model/1
//create m-object DaVinciCode//
product_dim/model/Book/model/2
//create m-object HarryPotter//
product_dim/model/Book/model/2
//update information of create m-object DaVinciCode//
product_dim/author/0
//update information of create m-object HarryPotter//
product_dim/author/0
```

Listing 5.10: Generated keys during optimization

Semantic validations

Before the translation of a SQL($\mathcal{M}$) statement starts, semantic validations can be done. For the SQL($\mathcal{M}$) interpreter semantic checks are implemented for the *CreateMultilevelObject* statement. This was merely done as a proof of concept as it was not part of the overall objective. The validations are implemented inside the class *SemanticChecker*. Altogether it provides three different kinds of checks.

First, the testing for loops inside the user-defined level hierarchy. The individual levels defining the level hierarchy must be hierarchically structured. The relations between the individual levels must be linear, otherwise it comes to data inconsistency at the date warehouse.

The second check is more a auto completion for incomplete level hierarchies. If an user-defined level hierarchy is incomplete the implemented algorithm auto completes the existing holes of the level hierarchy. Thereby users are able to send *CreateMultilevelObject* statements in abbreviated form, as long as the multilevel object inherits a level hierarchy from its parent.

The third check is the final validation of the level hierarchy consistency. Thereby is verified that the defined level hierarchy does not contradict already existing level hierarchies. Afterwards the translation of the statement can start. When an error is found during one of the different validations, an exception is thrown and the whole process is terminated.

Translation and execution

After the optimization, the translation process uses the information inside the abstract syntax tree and generates the corresponding equivalent in PL/SQL, usable by the hetero-homogenous data warehouse. The starting point of the translation is the *Translator* class. Its purpose is to identify the received statement types and start the respective translation process. The actual translation is done with the help of specialized translation classes. For each data definition (CREATE/ALTER/DROP) and data manipulation (UPDATE) statement an associated translation class exists. Each class contains the implementation for translating the corresponding statement type. For data query statements a separate class called *QueryTranslator* exists. It is called by the *Translator* class when query statements are found. For the *QueryTranslator* class also additional classes exists implementing specialized translation features.

As mentioned, for translating the data definition and data manipulation statements, specific classes were developed. Thereby each statement type has its own translation class, which has the same name as the corresponding statement type. E.g. for generating the PL/SQL code to create a multilevel object the class *CreateMultilevelObject* is used. All these translation classes derive from the superclass *Statement*. Figure 5.4 shows all existing translation subclasses under the superclass *Statement*.

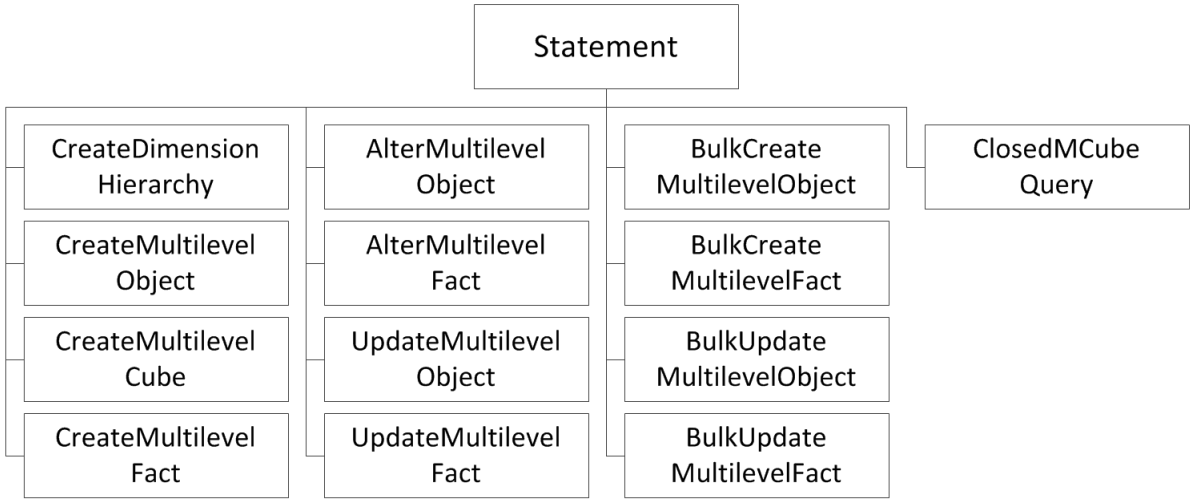


Figure 5.4.: The superclass *Statement* and derived subclasses

In Figure 5.4 no Java classes are listed for translating DROP statements. The SQL($\mathcal{M}$) language and grammar include four different DROP statements (see Chapter 4.1), but the corresponding functionality in the hetero-homogeneous data warehouse prototype is not yet fully implemented, since the data warehouse is periodically updated and typically data are not deleted. The focus of the SQL($\mathcal{M}$) implementation lies on the integration and querying of data. The translation classes for the DROP statements exist but only contain an implementation in beta version.

All implemented subclasses inherit the methods defined at the superclass. The method responsible for the translation process is the method named *translate*. It is defined as a *public abstract* method. This way it is made sure that every translation class provides this method. The purpose of the method is to trigger the translation process and return the executable PL/SQL code. Internally the *translate* methods are designed to call in

the proper order the methods implemented at the subclass. As the translation processes differ between SQL($\mathcal{M}$) statement types, so differ the structures of the different translation classes. The different phases of a translation process are broken down into various methods. This is mainly done for improving the readability of the Java code, but is also done for technical reasons. One reason is reusability, when recursive invocation is needed or recurring optional usage at different moments during the translation process.

Besides the presented translation subclasses two further translation classes exist. These contain translation functionality needed by multiple translation subclasses and are therefore encapsulated in separate classes. The classes are named *MultilevelObjectAttribute* and *MultilevelFactMeasure*. The class *MultilevelObjectAttribute* translates the addition of attributes to m-objects and the assignment of values to them. The class *MultilevelFactMeasure* provides the same functionality for measures and m-facts.

The following listings compare SQL($\mathcal{M}$) statements and the corresponding PL/SQL code created by the translation process.

```
CREATE DIMENSION HIERARCHY product_dim;

CREATE MULTILEVEL OBJECT Product IN DIMENSION product_dim
AT LEVEL top
HIERARCHY (
   LEVEL top CHILD OF NULL,
   LEVEL category CHILD OF top,
   LEVEL model CHILD OF category
);

CREATE MULTILEVEL OBJECT Book IN DIMENSION product_dim
AT LEVEL category UNDER Product
HIERARCHY (
   LEVEL category CHILD OF top,
   LEVEL model CHILD OF category (
      author VARCHAR2(40)
   )
);

CREATE MULTILEVEL OBJECT DaVinciCode IN DIMENSION product_dim
AT LEVEL model UNDER Book
SET author = 'Brown';

CREATE MULTILEVEL OBJECT HarryPotter IN DIMENSION product_dim
AT LEVEL model UNDER Book
SET author = 'Rowling';
```

Listing 5.11: SQL($\mathcal{M}$) translation input

Listing 5.11 shows example SQL($\mathcal{M}$) input, which consists of the same SQL($\mathcal{M}$) statements used in the previous examples. The example contains a *CreateDimensionHierarchy* statement defining the dimension *product_dim*. After that four m-objects are stated by useing *CreateMultilevelObject* statements. At first the root m-object *Product* is defined

for the dimension *product_dim*. Then follows the creation of the m-object *Book* at the level *category*. This statement also adds the attribute *author* at level *model* with data type VARCHAR2. At the end two m-objects, namely *DaVinciCode* and *HarryPotter*, are defined at level *model*, setting the attribute *author*. Listing 5.12 shows the generated PL/SQL code of SQL($\mathcal{M}$) interpreter. Comparing those two listings one can clearly see that with SQL($\mathcal{M}$) the same amount of information is defined with less code than in PL/SQL. Further the SQL($\mathcal{M}$) statements are easier to understand and comprehend than the PL/SQL code. As a consequence users save time by using SQL($\mathcal{M}$) due to pure code reduction and improved usability.

```
DECLARE
  dim REF dimension_ty;
BEGIN
  dim := dimension.create_dimension('product_dim');
END;

DECLARE
  d dimension_ty;
  mobject_ref REF mobject_ty;
  mobject mobject_ty;
  parents mobject_trty;
  levelhierarchy level_hierarchy_tty;
  parent_onames names_tty := names_tty(NULL);
BEGIN
  SELECT VALUE (dim) INTO d
  FROM dimensions dim
  WHERE dim.dname = 'product_dim';
  levelhierarchy := level_hierarchy_tty(
                level_hierarchy_ty('top', NULL),
                level_hierarchy_ty('category', 'top'),
                level_hierarchy_ty('model', 'category'));
  mobject_ref := d.create_mobject('Product', 'top', NULL,
     levelhierarchy);
END;

DECLARE
  d dimension_ty;
  mobject_ref REF mobject_ty;
  mobject mobject_ty;
  parents mobject_trty;
  levelhierarchy level_hierarchy_tty;
  parent_onames names_tty := names_tty('Product');
BEGIN
  SELECT VALUE (dim) INTO d
  FROM dimensions dim
  WHERE dim.dname = 'product_dim';
  parents := mobject_trty();
  parents.extend;
  parents(parents.LAST) := d.get_mobject_ref('Product');
```

```
  levelhierarchy := level_hierarchy_tty(
                level_hierarchy_ty('category', 'top'),
                level_hierarchy_ty('model', 'category'));
  mobject_ref := d.create_mobject('Book', 'category', parents,
     levelhierarchy);
  utl_ref.select_object(mobject_ref, mobject);
  mobject.add_attribute('author', 'model', 'VARCHAR2(40)');
END;

DECLARE
  d dimension_ty;
  parents mobject_trty;
  levelhierarchy level_hierarchy_tty;
  onames names_tty;
BEGIN
  SELECT VALUE (dim) INTO d
  FROM dimensions dim
  WHERE dim.dname = 'product_dim';
  onames := names_tty('DaVinciCode', 'HarryPotter');
  parents := mobject_trty();
  parents.extend;
  parents(parents.LAST) := d.get_mobject_ref('Book');
  d.bulk_create_mobject(onames, 'model', parents, levelhierarchy);
END;

DECLARE
  d dimension_ty;
BEGIN
  SELECT VALUE (dim) INTO d
  FROM dimensions dim
  WHERE dim.dname = 'product_dim';
  d.bulk_set_attribute('author',
                       mobject_value_tty(
                           mobject_value_ty(
                               'DaVinciCode',
                                ANYDATA.convertVarchar2('Brown')),
                           mobject_value_ty(
                               'HarryPotter',
                                ANYDATA.convertVarchar2('Rowling'))
                           ));
END;
```

Listing 5.12: PL/SQL translation output

As previously mentioned, for translating data query statements a separate class called *QueryTranslator* exists. The *QueryTranslator* class determines the kind of query statement it received and starts the corresponding translation process. The differentiation is made between query statements containing a sub-query, which is called ClosedMCubeQuery, or not. The sub-query from the SQL($\mathcal{M}$) language can be compared with the

sub-select statements from SQL. It allows the user to nest arbitrary *ClosedMCubeQuery* within each other. For the translation of a *ClosedMCubeQuery* a separate class called *ClosedMCubeQuery* is implemented. As arbitrary *ClosedMCubeQuery* can be related with each other, the class was designed to be able to call itself recursive till all *ClosedMCube-Query* are processed.

Each *ClosedMCubeQuery* of a SQL($\mathcal{M}$) query statement is processed separately. Thereby the translation process starts with the most inner sub-query and works its way up to the most outer query. In the data warehouse the queries are evaluated with a *queryview* object. The *queryview* loads the selected cube and processes the committed operations, such as slice, dice and projection. After all operations of all sub-queries are done, the remaining data is saved into a temporary table created by the *queryview* during the final rollup operation. The SQL($\mathcal{M}$) interpreter then queries the result from the table and returns it to the user.
The following listings compare SQL($\mathcal{M}$) query statements and the corresponding PL/SQL code.

```
SELECT *
FROM ROLLUP[product_dim.category,
            time_dim.year,
            location_dim.country]
   (SELECT MULTILEVEL revenue
    FROM DICE[Book, Year2010, Salzburg] (sales_cube) s
    WHERE s.product_dim.model.costs < 35);
```

Listing 5.13: SQL($\mathcal{M}$) query input

Listing 5.13 displays a SQL($\mathcal{M}$) query input. The input example covers all the basic functionality provided by the hetero-homogeneous data warehouse (rollup, dice, projection, slice). At first the ROLLUP clause defines the degree of abstraction to which the result is calculated. Then follows a sub-query (*ClosedMCubeQuery*) where the queried measures are reduced (projection). After the FROM clause the dice functionality reduces the selected m-cube the to indicated m-objects. At the WHERE clause of the sub-query the result is further specified by a slice expression.

```
DECLARE
   m_cube mcube_c000000001_ty;
   mrel mrel_c000000001_ty;
   dim_ref REF dimension_ty;
   queryview queryview_c000000001_ty;
   pred_0 slice_predicate_ty;
   pred_1 slice_predicate_ty;
   pred_2 slice_predicate_ty;
BEGIN
   SELECT TREAT(VALUE(mc) AS mcube_c000000001_ty) INTO m_cube
   FROM mcubes mc
   WHERE mc.cname = 'sales_cube';
   queryview := m_cube.new_queryview;

   SELECT REF(d) INTO dim_ref
   FROM dimensions d
```

```
   WHERE d.dname = 'product_dim';
   pred_0:=
      slice_predicate_ty(dim_ref, 'model');
   pred_0.add_expression('price', '<',
                         ANYDATA.convertNumber(35));

   queryview := queryview.slice(pred_0, NULL, NULL);
   queryview := queryview.project(names_tty('revenue'));
   queryview := queryview.dice('Book', 'Year2010', 'Salzburg');
   queryview := queryview.evaluate;

   queryview.rollup('temp_Output_0', FALSE,
                    'category', 'year', 'city');
END;
```

Listing 5.14: PL/SQL query output

Listing 5.14 shows how the PL/SQL code looks like for the just described SQL($\mathcal{M}$) query and illustrates the usage of the functionality provided by the hetero-homogeneous data warehouse.

When comparing the two query possibilities the advantages of SQL($\mathcal{M}$) are evident, since it exceeds the PL/SQL code in terms of usability and comprehensibility.

DataDictionary

For generating executable PL/SQL code, sometimes, information of the data warehouse about already existing data is needed. Furthermore the names of the dynamically generated object types are needed for the translation process. To reduce the effort of querying the same information multiple times the class *DataDictionary* is implemented. It is an intermediate store for all the information which could be needed again during the translation process.

Beside the queried data of the data warehouse, the *DataDictionary* also stores information about dynamic created entities by the interpreter during the translation process. An example are the names of temporary database tables, which should be deleted after the translation process.

The class *DataDictionary* is implemented by applying the *singleton* pattern. As a consequence only one instance of this object type can exist during runtime. If you try to instantiate more then one object of type *DataDictionary*, starting from the second instantiation the object constructor always returns the reference of the first object. This restriction is implemented so that data consistency can be guaranteed. Since only one *DataDictionary* object can exist all information is gathered at one point and is thereby always up to date, which is important to generate valid code.

DataAccess

The class *DataAccess* provides the functionality to connect to the database were the data warehouse is situated, send queries and receive their results. The connection is established through the Oracle server-side internal driver. This is a specialized JDBC driver, granting fast access to the database [Ora09, 1-17]. The server-side internal driver runs within the default session of the executing user. Thereby no further authentication is

needed to create a database connection as the application is already connected via the users database session [Ora09, 2-1 et seq.]. Listing 5.15 shows how to open a database connection in Java by using the Oracle internal driver.

```
private Connection getConnection(){
   Connection cn;
      try {
         Class.forName("oracle.jdbc.driver.OracleDriver");
         cn = DriverManager.getConnection("jdbc:default:connection:");
      } catch (Exception e) {
         /** Error handling **/
      }
      return cn;
}
```

Listing 5.15: Using the interpreter API in Java applications

Like *DataDictionary*, the class *DataAccess* is implemented by using the *singleton* pattern. In case of the *DataAccess* class this was done for controlling the number of open connections to the database. As the *DataAccess* class can only establish one connection at a time and only one *DataAccess* object can be instantiated, the number of connections stays constant at one.

Interpreter API

To be able to communicate with the interpreter an application programming interface (API) is implemented, providing access to the interpreter functionality. In the case of the SQL($\mathcal{M}$) interpreter the interface consists of two parts. One situated at database level and the other at interpreter level. The interface at database level is explained in detail in Section 5.2.

The interface part at the interpreter level is a Java method called "sqlmQuery". Listing 5.16 shows the design of the method. As input the method receives data in form of SQL data type CLOB (Character Large Object) and returns a Java ResultSet to the invoking SQL function.

Inside the method the input data is transformed from SQL data type CLOB to a Java object of type StringReader. This is done as the parser functionality generated by the JavaCC framework needs as input the data in form of a Java StringReader object. After the transformation of the input the three main processes of the SQL($\mathcal{M}$) interpreter are triggered in chronological order:

- Parsing (via class *SQLMParser*)
- Optimization (via class *Optimizer*)
- Translation and execution (via class *Translator*)

```
public static ResultSet sqlmQuery(oracle.sql.CLOB clob_input){
   String input = clob_input.getSubString(1, (int)clob_input.length());
   StringReader reader_input = new StringReader(input);
   if(first){
      new SQLMParser(reader_input);
      first = false;
   }else{
      SQLMParser.ReInit(reader_input);
   }
   ASTSQLMDocument document = SQLMParser.Start();
   Optimizer optimizer = new Optimizer();
   optimizer.run(document);
   Translator translator = new Translator();
   return translator.run(document);
}
```

Listing 5.16: Interpreter API definition

6. Conclusion

In this thesis we presented the SQL($\mathcal{M}$) data definition and query language, implemented to provide easy-to-use access to hetero-homogeneous data warehouses. Therefore the SQL($\mathcal{M}$) interpreter was developed as a Java stored procedure for reading SQL($\mathcal{M}$) statements. The interpreter translates the SQL($\mathcal{M}$) input into PL/SQL code and executes the generated code. Further the overall system design and underlying process were presented by illustrating the interactions between the different system pieces and participants. The proof-of-concept implementation of the SQL($\mathcal{M}$) interpreter suggests that SQL($\mathcal{M}$) provides increased usability for the end user compared to accessing the data warehouse using the PL/SQL functions directly. The SQL($\mathcal{M}$) interpreter and editor provide another option for working with hetero-homogeneous data warehouses apart from the existing OLAP client [Rot15], [SRN15].

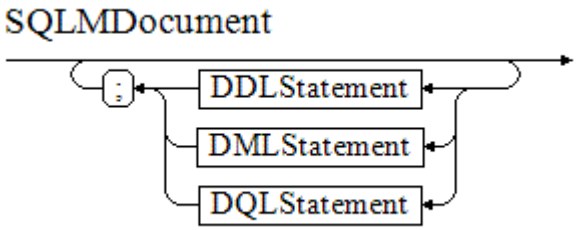
A. SQL($\mathcal{M}$) railroad description

This section presents the SQL($\mathcal{M}$) grammar in form of railroad diagrams. Railroad diagrams (also called syntax diagrams) are a visual way to illustrate context-free grammars. Here they are used to visualize the structure of SQL($\mathcal{M}$) and show how to use the provided functionality. Every diagram has a start and an end point, describing valid sequences of symbols. Thereby different graphical shapes are used, which represent the different grammar components. The following list shows the used graphical notation for the presented diagrams inside this section:

- rectangle: nonterminal symbol
- oval: terminal symbol
- arrow: start/end point, indicates reading direction

The diagrams were generated with the help of a tool called EBNF Visualizer[1], written by Stefan Schoergenhumer and Markus Dopler, supported by Hanspeter Moessenboeck, of the University of Linz.

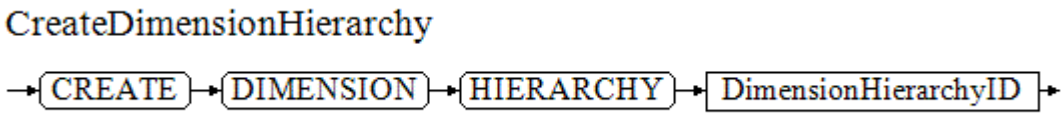
After the creation, some of the diagrams were slightly manually reworked, to fit in this document. The following diagrams are sorted by their dependencies, this shall benefit the reading flow and the comprehensibility. All starts at the root node of the SQL($\mathcal{M}$) grammar, where the three fundamental statement categories of SQL($\mathcal{M}$) are distinguished. These three categories are *data definition*, *data manipulation* and *data query* statements.



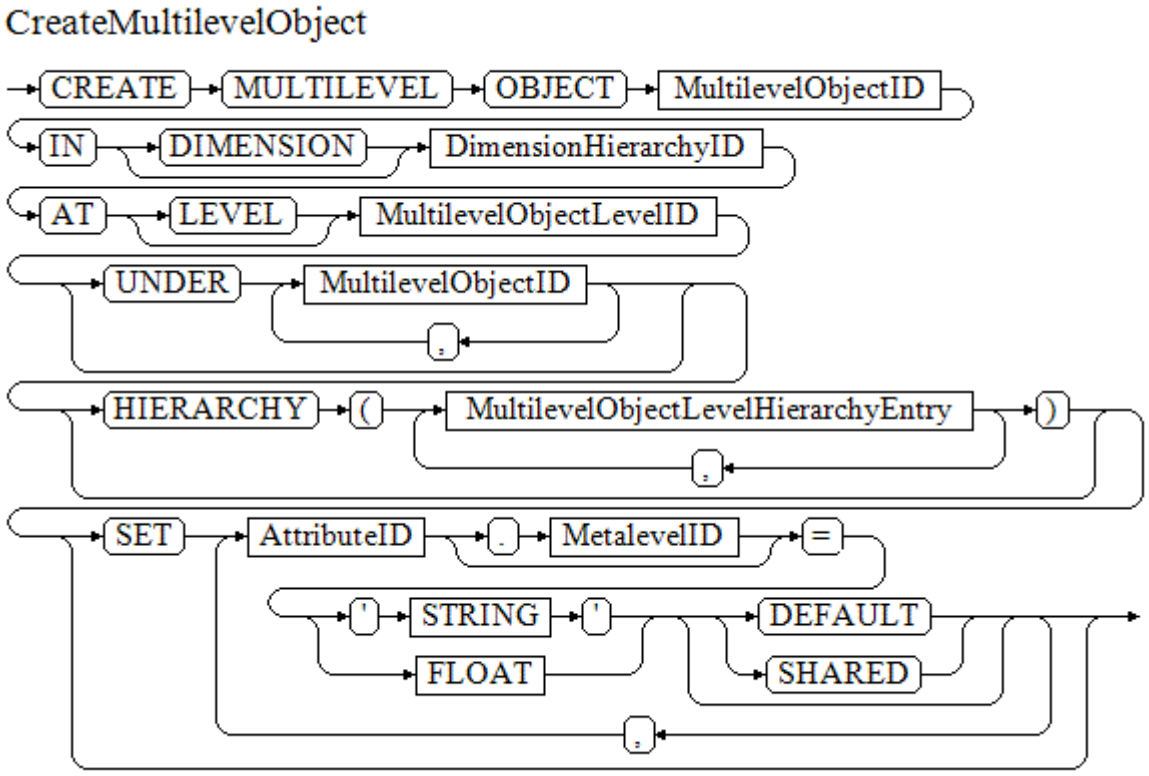
An *SQLDocument* may contain any number of statements, separated by semicolon. The current implementation of SQL($\mathcal{M}$) interpreter and editor allow only the execution and display of result of one query statement at a time.

[1] http://dotnet.jku.at/applications/visualizer/

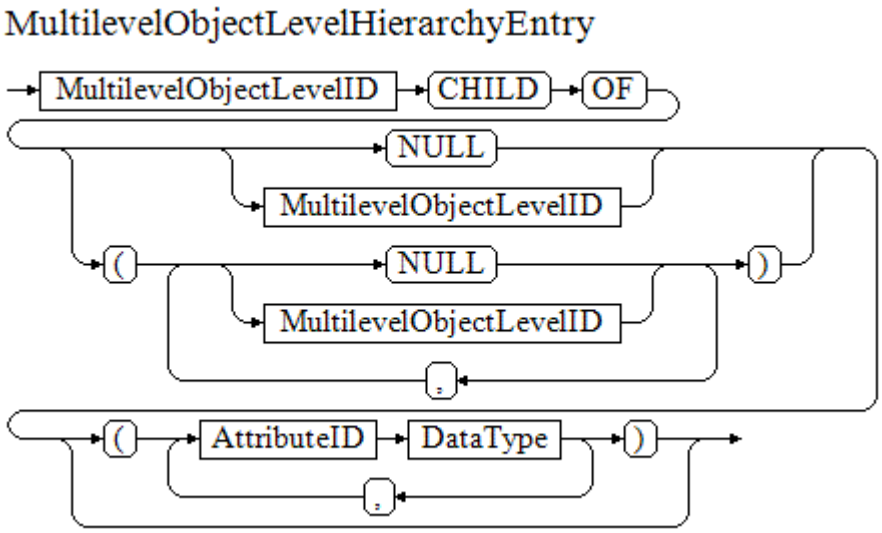
Data definition statements



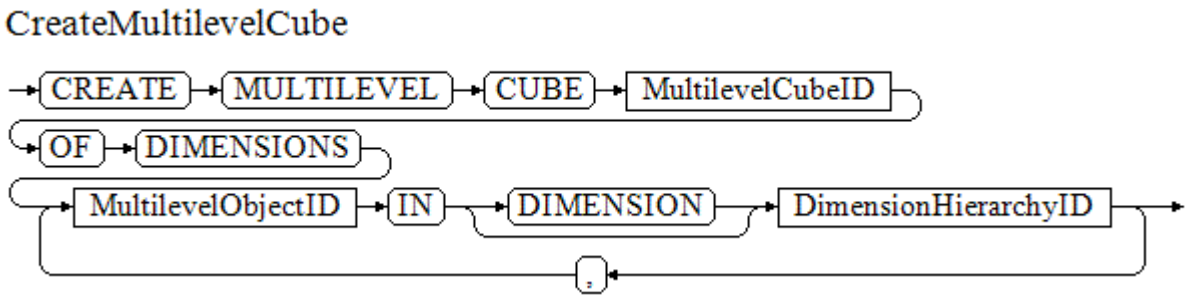
The *DimensionHierarchyID*, i.e. the name of the dimension, is of data type *String*.



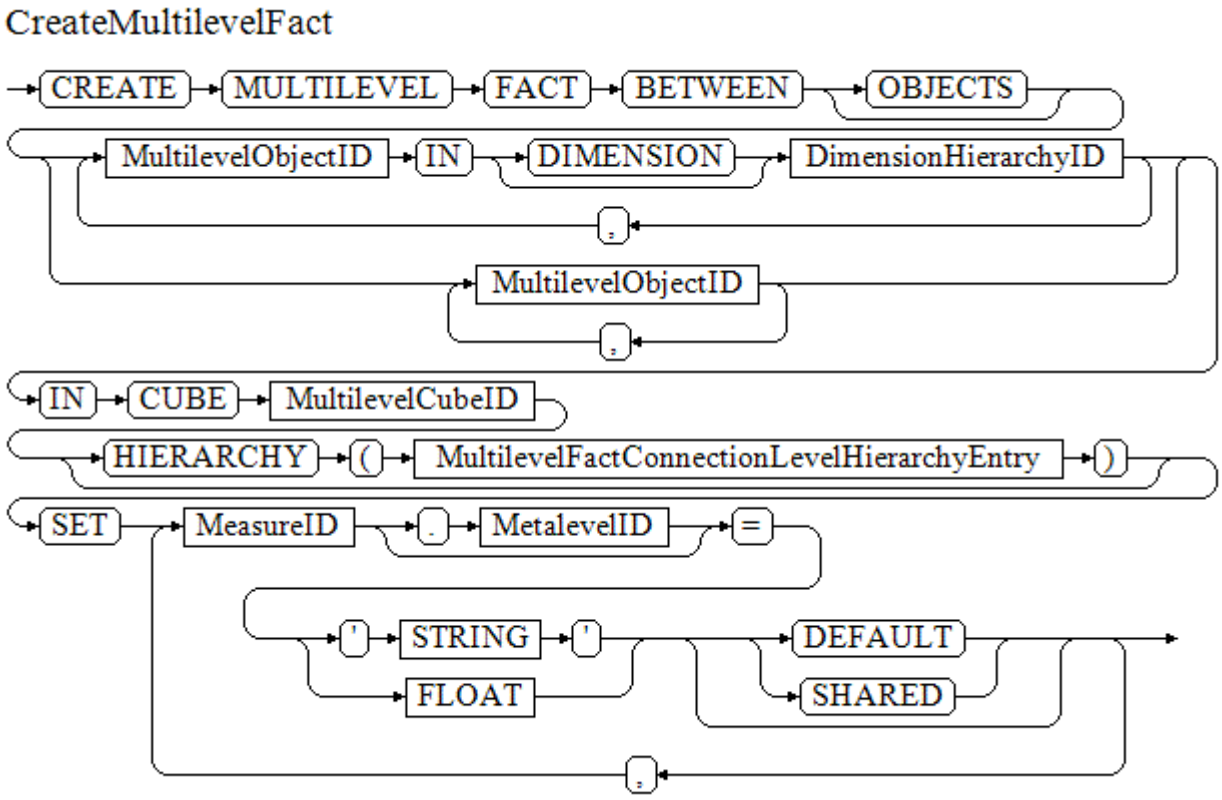
The *MultilevelObjectLevelID* and the *AttributeID* are of type *String*. How to define a level hierarchy is described in detail in the next diagram.



Each entry represents a level inside the coresponding m-object. The name of the level is of type *String*. After the name of the level the parent levels are defined. If there is more than one parent object, the enumeration has to be within brackets. After the definition of a level, attributes for this level can be stated.



For the creation of a m-cube, the name for the cube is needed and the *Dimension-HierarchyIDs* which the cube consists. For every dimension a m-object must be definied. This m-objects represent the aggregation limits of the corresponding dimensions inside the m-cube. Both *DimensionHierarchyIDs* and *MultilevelObjectID* are of type *String*.

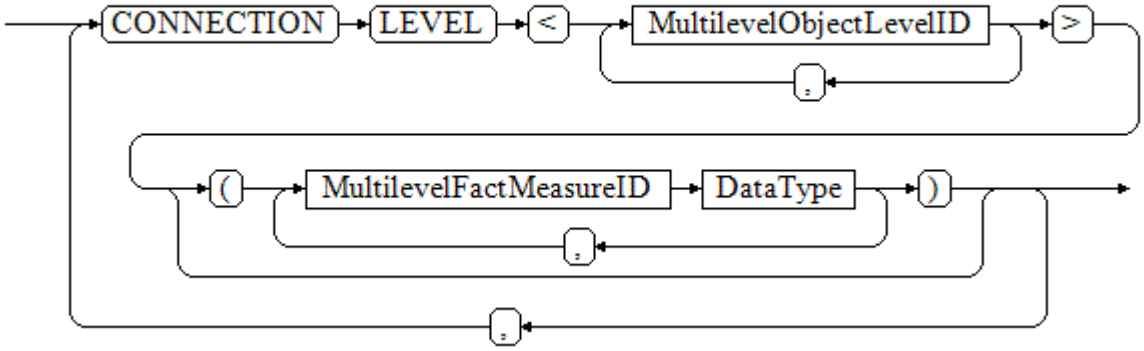


To create a m-fact, the m-fact's coordinate inside the m-cube must be declared. This *MultilevelFactID* can be assigned in two ways.

1. Qualified
2. Unqualified

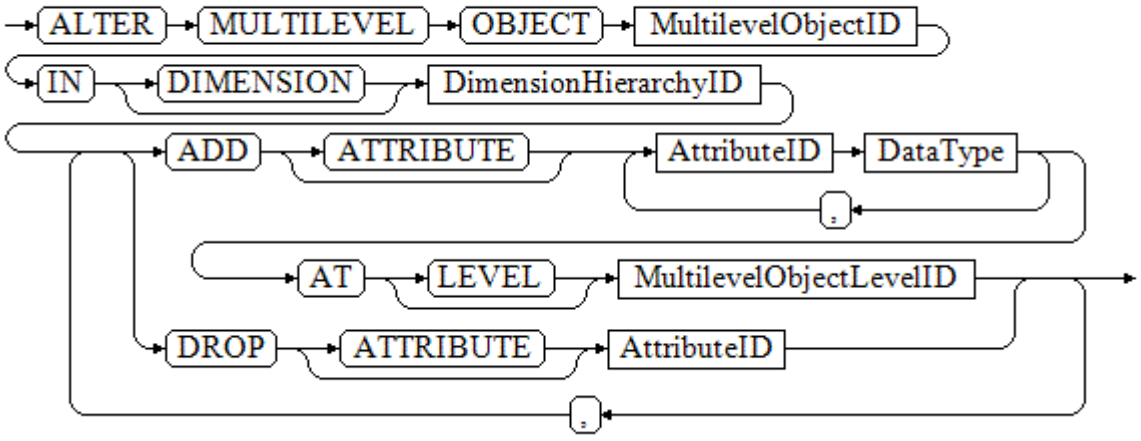
Qualified means that the m-objects and their corresponding dimensions are defined for the *MultilevelFactID*. In this case the m-object order must not be valid with the cube coordinate of the m-cube the m-fact is created for. The interpreter can check the order and correct it if needed. When the *MultilevelFactID* is specified unqualified, i.e. without *DimensionHierarchyIDs*, then the user must procure that the m-object order matches with the cube coordinate.

MultilevelFactConnectionLevelHierarchy

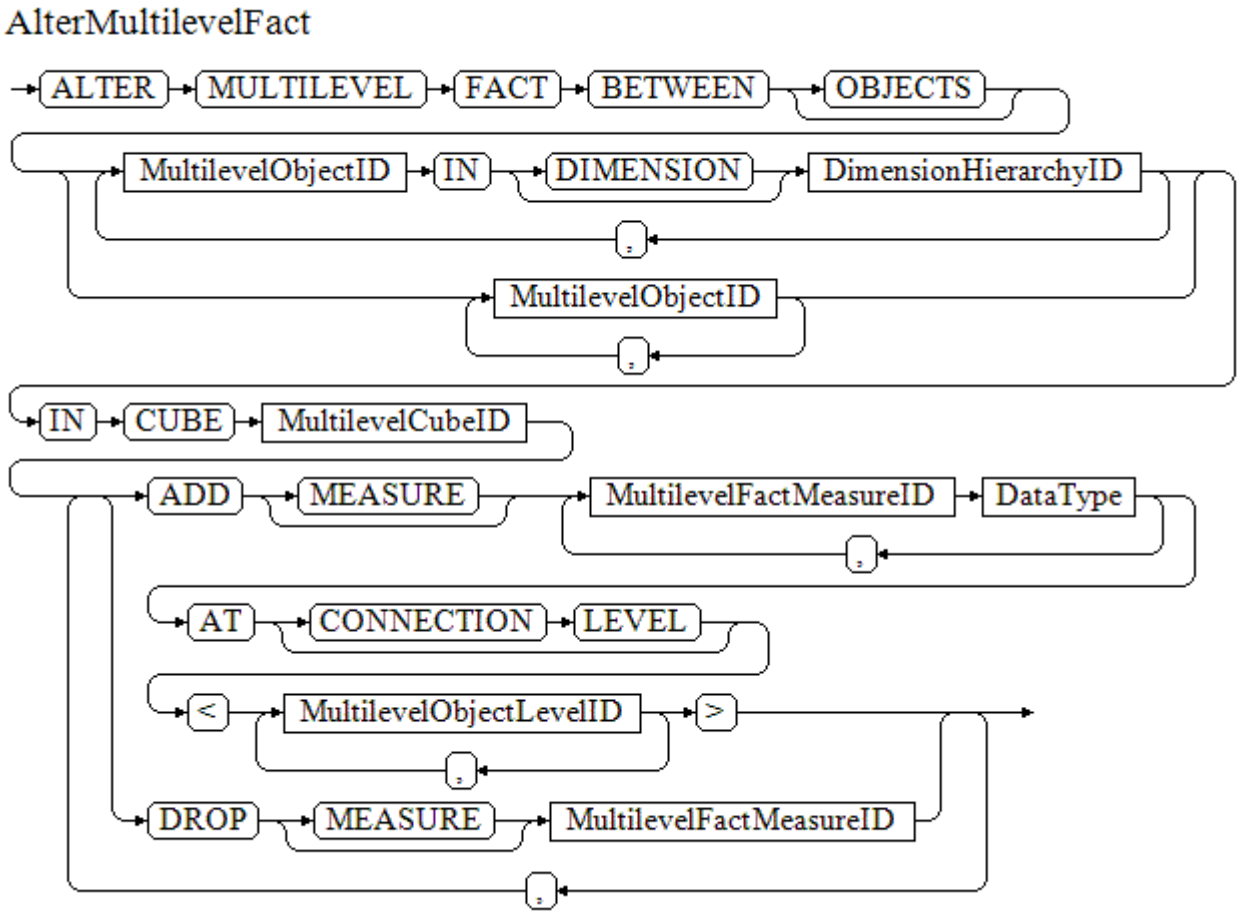


In this part of the statement new measures can be defined. First the connection level hierarchy of the new measure has to be declared. Thereby the dimensional order of the *LevelIDs* has to match with the dimensional order of the corresponding m-cube's cube coordinate. Afterwards the measure can be specified by a *MeasureID* of type *String* and a data tpye (E.g. VARCHAR2).

AlterMultilevelObject



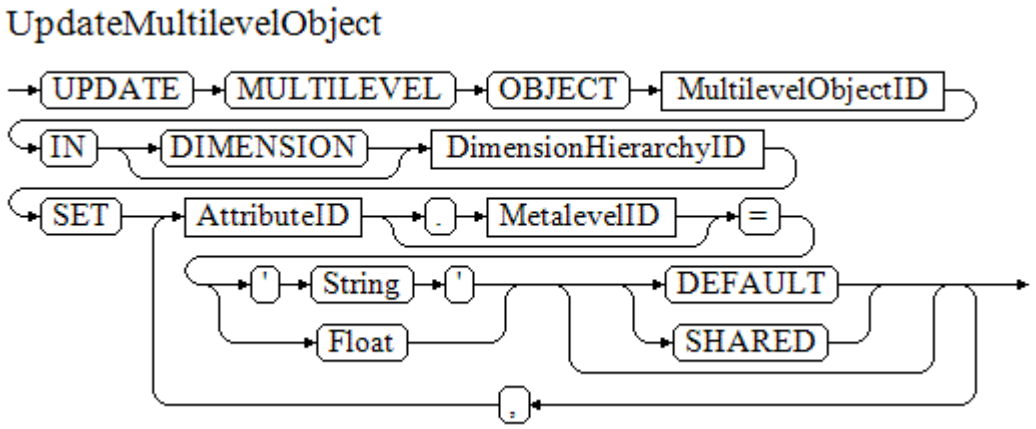
To alter attributes of a m-object, first the m-object has to be declared with the *MultilevelObjectID* and the corresponding dimensionID, both of type *String*. After that arbitrary add and drop declarations can be assigned.



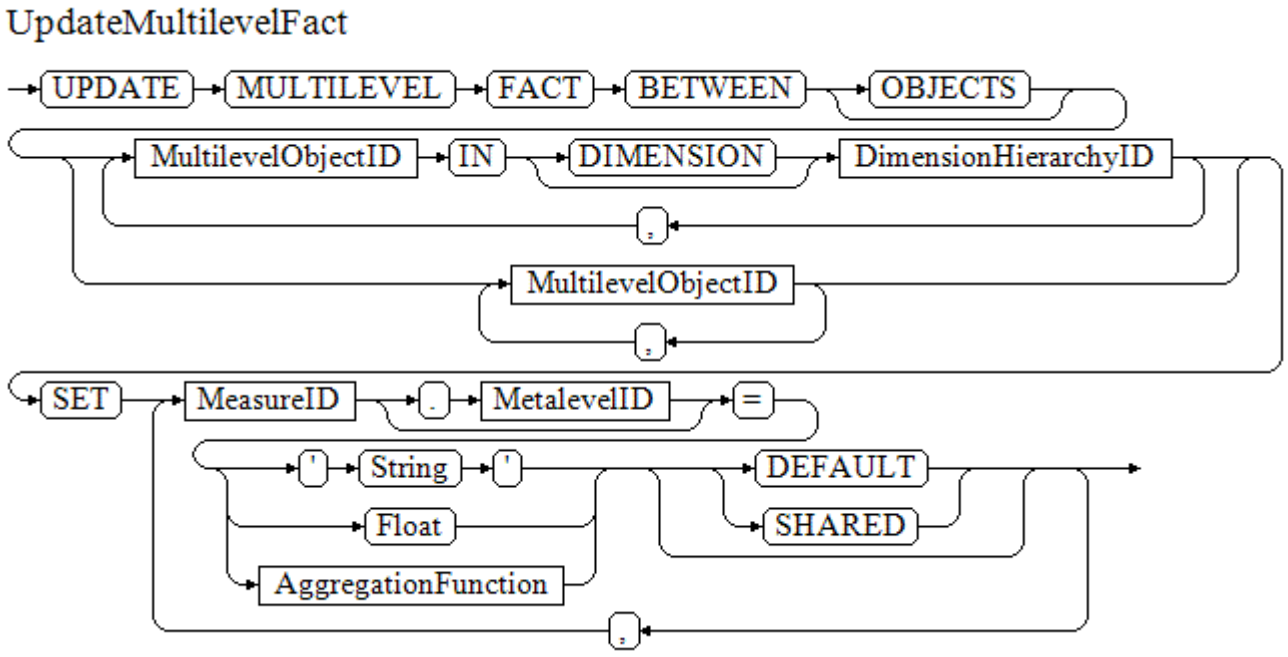
For editing a m-fact, the *MultilevelFactID* has to be delared first. This can be done in two ways. Qualified and Unqualified, as already mentioned at the CreateMultilevelFact diagram. Afterwards multiple add and drop declarations can be defined.

Data manipulation statements

The next diagrams illustrate the UPDATE functionality, which covers how to set values for attributes and measures.



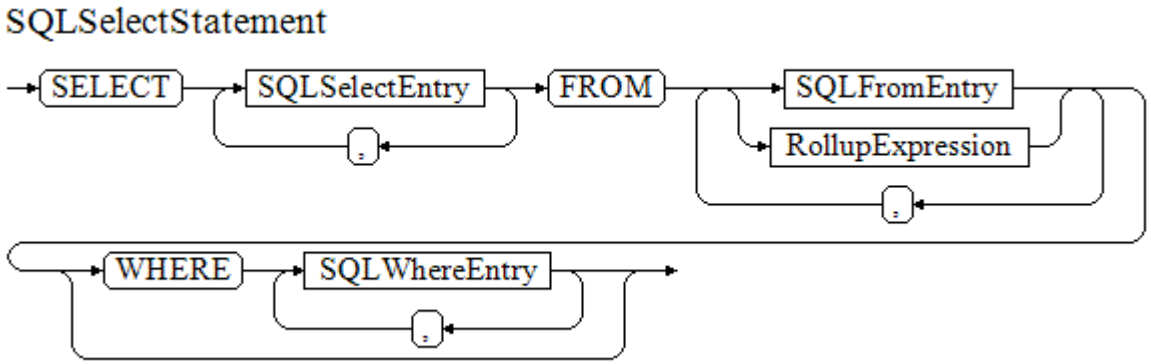
To set an attribute value, first the *MultilevelObjectID* and the corresponding *DimensionHierarchyID* have to be declared, both of type *String*. Then the values for arbitrary attributes can be assigned. The attributeID is of type *String*, whereas the attribute value types can differ.



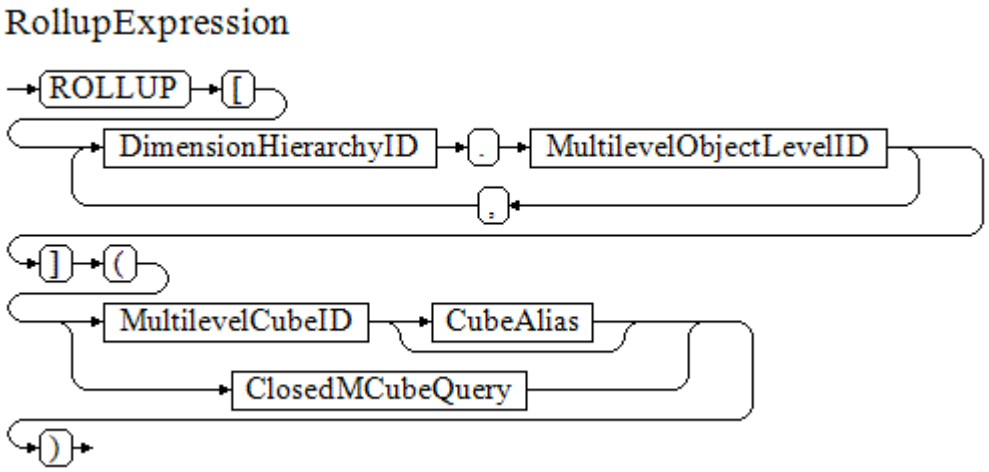
To set measure values, first the declaration of the *MultilevelFactID* is needed. This can be done in two ways, qualified and unqualified, as already mentioned at the CreateMultilevelFact diagram. Values can be set for arbitrary measures. The *MeasureID* is thereby defined as *String*, for the measure values different kinds of types are possible.

Data query statements

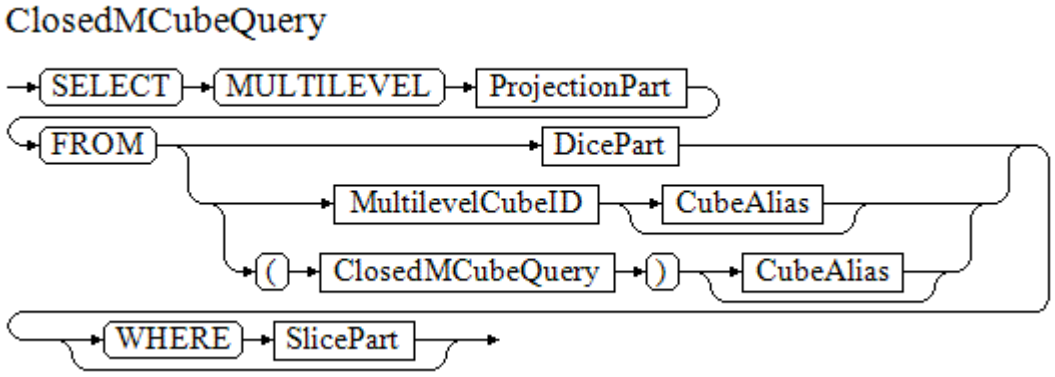
Now the structure of query statements is shown in detail. The presented diagrams illustrate how the different query operators are addressed correctly.



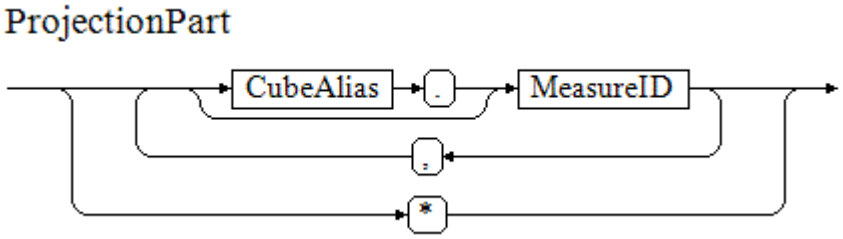
The SQL($\mathcal{M}$) query functionality is implemented into the standard *SQL SELECT statement*. Thereby the known SQL commands for this statement type are available. Through the ROLLUP operator the mulitlevel m-cubes can be accessed in a relational setting. It is situated inside the SQL FROM clause.



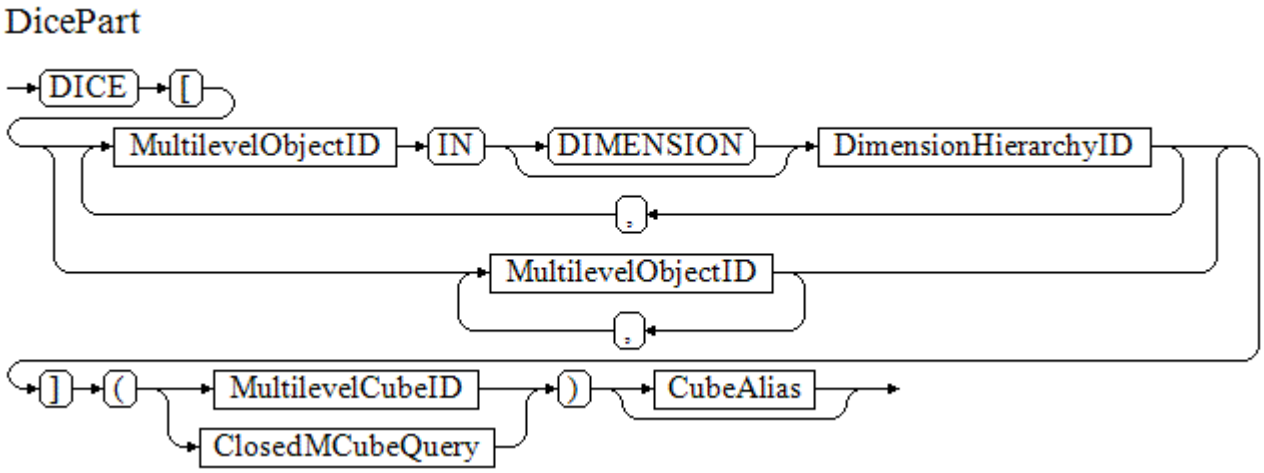
The ROLLUP functionality transforms the invoked m-cube as specified by the rollup-granularity level. Two different kinds of m-cube inputs are supported. A m-cube without restriction, or a subcube by using a closed m-cube query.



The *ClosedMCubeQuery* provides the three data aggregation operations supported by SQL($\mathcal{M}$) as already specified by the closed m-cube query algebra [NST10]. The three operations are explained in detail in the following diagrams.

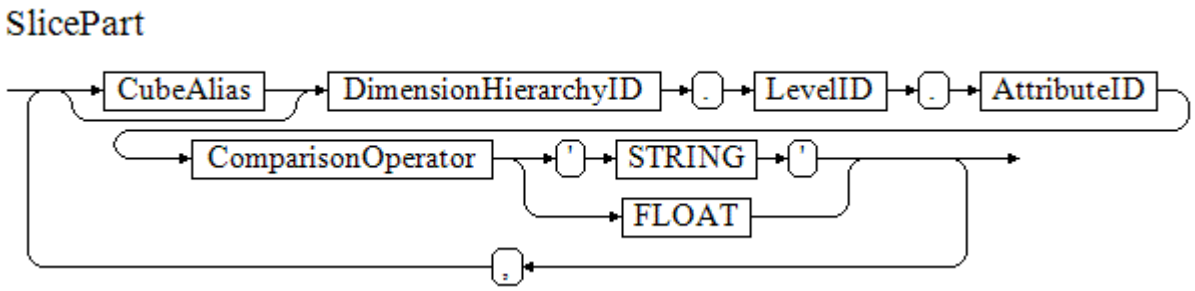


With the use of the *projection* functionality, a list of to be considered measures can be defined. All other available measures are ignored. Measures are declared by specifying the measure name, which is of type *String*. With the use of the asterisk all measures of a m-cube are selected.



The DICE operator narrows down the considered base data to the m-facts under the specified dice coordinate. This can be done in two different ways. One is to only declare the m-objects using their ID's of type *String*. Then the chronological order of the committed m-objects has to be valid with the m-cube's cube coordinate. The second way

is to declare the m-objects with their corresponding dimensions. Then the interpreter automatically checks the stated order and adjusts it when needed.



The *slice* function restricts the data are considered, by defining selection criteria on attributes. A slice expression consists of an attribute, a comparison operator and a value.

B. Test plan

The following test plan is designed as a "black-box" test. This means the functionality of the interpreter is tested by giving the interpreter all kinds of different input statements and evaluating the produced database status respectively output. As a first step the to be tested functionality of the interpreter will be specified. This includes:

- statements which a user can insert
- optimization functionality (automatically generated BULK statements)
- execution order

After the definition of the test scope a SQL($\mathcal{M}$) test input is presented. The input data is designed to cover all test definitions of the defined scope.

B.1. Test scope

Now as the first step the to be tested statements and interpreter functionalities are defined. For every defined entity the mandatory and optional data is listed. After that for each entity a set of tests is specified. A set covers the possibilities of different types of a defined entity. In the following lists every test is categorised and numbered for later reference in the test data.

Create Dimension Hierarchy (CDH)

Mandatory data:
- dimension hierarchy name

Code	Test description
CDH	Elementary statement consisting of mandatory data

Table B.1.: Test set for 'Create Dimension Hierarchy' statement

Create Multilevel Object (CMO)

Mandatory data:

- multilevel object name
- dimension hierarchy name
- level name

Optional data:
- level hierarchy (only exceptions is the top level multilevel object of a each dimension hierarchy ->there the level hierarchy is obligatory)
- attribute data

Code	Test description
CMO1	Elementary statement which consists only of mandatory data
CMO2a	Statement which expands the level hierarchy (hierarchy is completely defined by user)
CMO2b	Statement which expands the level hierarchy (only the changes are defined by user -> the residual levels must be filled-up by the interpreter)
CMO2c	Statement which expands the level hierarchy by using multiple inheritance
CMO3	Statement contains attribute data
CMO4	Statement adds a new attribute through the level hierarchy

Table B.2.: Test set for 'Create Multilevel Object' statement

Create Multilevel Cube (CMC)

Mandatory data:
- multilevel cube name
- dimension hierarchy names defining the edges of the multilevel cube

Code	Test description
CMC	Elementary statement consisting of mandatory data

Table B.3.: Test set for 'Create Multilevel Cube' statement

Create Multilevel Fact (CMF)

Mandatory data:

- multilevel cube name
- multilevel fact cube coordinate

Optional data:
- connection level hierarchy (contains connection levels and associated measures)
- measure data

Code	Test description
CMF1a	Basic statement which consists only of mandatory data and uses qualified cube id
CMF1b	Basic statement which uses an unqualified cube id
CMF2a	Statement which creates a connection level hierarchy
CMF2b	Statement which creates a connection level hierarchy and defines measures in it
CMF3a	Statement contains measure data
CMF3b	Statement contains measure meta data

Table B.4.: Test set for 'Create Multilevel Fact' statement

Alter Multilevel Object (AMO)

Mandatory data:

- multilevel object name
- dimension name of the altered multilevel object
- data for adding an attribute:
  attribute name
  data type of the attribute
  level name where the attribute should be added
- data for deleting an attribute:
  attribute name

Code	Test description
AMO1	Statement adding an attribute
AMO2	Statement dropping an attribute
AMO3	Statement which adds and drops attributes

Table B.5.: Test set for 'Alter Multilevel Object' statement

Alter Multilevel Fact (AMF)

Mandatory data:

- multilevel cube name
- multilevel fact cube coordinate
- data for adding a measure:
  measure name
  data type of the measure
  connection level where the measure should be added
- data for deleting a measure:
  measure name

Code	Test description
AMF1	Statement adding a measure
AMF2	Statement dropping a measure
AMF3	Statement which adds and drops measures

Table B.6.: Test set for 'Alter Multilevel Fact' statement

Update Multilevel Object (UMO)

Mandatory data:

- multilevel object name
- dimension hierarchy name
- attribute name

Optional data:

- meta level data

- attribute value

Code	Test description
UMO1	Statement updating an attribute
UMO2	Statement updating an attribute with meta-level data
UMO3	Statement updating multiple attributes

Table B.7.: Test set for 'Update Multilevel Object' statement

Update Multilevel Fact (UMF)

Mandatory data:
- multilevel fact cube coordinate
- multilevel cube name
- measure name
- measure value

Optional data:
- meta level data

Code	Test description
UMF1	Statement updating a measure
UMF2	Statement updating a measure with meta-level data
UMF3	Statement updating multiple measures

Table B.8.: Test set for 'Update Multilevel Fact' statement

Query Statements (QS)

Mandatory data:
- multilevel fact cube coordinate
- multilevel cube name

Optional data:
- multiple multilevel cubes
- subquery instead of a multilevel cube name

Code	Test description
QS1	Basic query with only the ROLLUP operator)
QS2	Query with multiple cubes selected
QS3	Query with closed m-cube query as input
QS4	Query using the projection functionality
QS5	Query using the dice functionality
QS6	Query using the slice functionality
QS7	Query using slice functionality multiple times
QS8	Query using the convert functionality

Table B.9.: Test set for query Statements

Optimization (BULK statements)

The execution order of statements can be important. For example, a Multilevel Object always must be created first before it can be altered or updated. The user is responsible that these dependencies between statements are correct. If during the optimization process statements are joined together, the dependencies between them and other concerned statements must still be fulfilled. Therefore, the interpreter includes functionalities for guaranteeing the validity of the execution order. Now follows the test specification of all possible Bulk statements and the corresponding execution order tests.

Bulk Create Multilevel Object (BCMO)

to match data for optimization:
- multilevel object parent name
- dimension hierarchy name
- level name where the multilevel object is situated

Code	Test description
BCMO1	Basic Bulk Create Multilevel Object Statement
BCMO2	Bulk Create Multilevel Object Statement with attribute data inside the assembled create statements

Table B.10.: Test set for 'Bulk Create Multilevel Object' statement

Bulk Create Multilevel Fact (BCMF)

to match data for optimization:
- multilevel fact cube coordinate
- multilevel cube name

Code	Test description
BCMF1	Basic Bulk Create Multilevel Fact Statement
BCMF2	Bulk Create Multilevel Fact Statement with measure data inside the assembled Create statements

Table B.11.: Test set for 'Bulk Create Multilevel Fact' statement

Bulk Update Multilevel Object (BUMO)

to match data for optimization:
- dimension hierarchy name
- attribute name

Code	Test description

BUMO	Bulk Update Multilevel Object Statement

Table B.12.: Test set for 'Bulk Update Multilevel Object' statement

Bulk Update Multilevel Fact (BUMF)

to match data for optimization:

- multilevel cube name
- measure name

Code	Test description
BUMF	Bulk Update Multilevel Fact Statement

Table B.13.: Test set for 'Bulk Update Multilevel Fact' statement

B.2. Test Data

Nr.	Code	SQL($\mathcal{M}$) statements
1	CDH	CREATE DIMENSION HIERARCHY product_dim; CREATE DIMENSION HIERARCHY time_dim; CREATE DIMENSION HIERARCHY location_dim; CREATE DIMENSION HIERARCHY quantity_dim;
2	CMO2a	CREATE MULTILEVEL OBJECT Quantity IN DIMENSION quantity_dim AT LEVEL top HIERARCHY (LEVEL top CHILD OF NULL, LEVEL quantity CHILD OF top, LEVEL unit CHILD OF quantity);
3	CMO2a	CREATE MULTILEVEL OBJECT Currency IN DIMENSION quantity_dim AT LEVEL quantity UNDER Quantity HIERARCHY (LEVEL quantity CHILD OF top, LEVEL unit CHILD OF quantity);
4	CMO2a	CREATE MULTILEVEL OBJECT Euro IN DIMENSION quantity_dim AT LEVEL unit UNDER Currency; CREATE MULTILEVEL OBJECT Dollar IN DIMENSION quantity_dim AT LEVEL unit UNDER Currency;

5	CMO2a	CREATE MULTILEVEL OBJECT Product IN DIMENSION product_dim AT LEVEL top HIERARCHY (LEVEL top CHILD OF NULL, LEVEL category CHILD OF top (categoryManager VARCHAR2(40)), LEVEL model CHILD OF category);
6	AMO1	ALTER MULTILEVEL OBJECT Product IN DIMENSION product_dim ADD ATTRIBUTE price NUMBER AT LEVEL model;
7	CMO2a CMO4	CREATE MULTILEVEL OBJECT Book IN DIMENSION product_dim AT LEVEL category UNDER Product HIERARCHY (LEVEL category CHILD OF top, LEVEL model CHILD OF category (author VARCHAR2(40)));
8	UMO1	UPDATE MULTILEVEL OBJECT Book IN DIMENSION product_dim SET categoryManager = 'MrBookMgr';
9	CMO2b CMO3	CREATE MULTILEVEL OBJECT Car IN DIMENSION product_dim AT LEVEL category UNDER Product HIERARCHY (LEVEL category CHILD OF top, LEVEL brand CHILD OF category(brandName VARCHAR2(40)), LEVEL model CHILD OF brand (maxSpeed NUMBER)) SET categoryManager = 'MrCarMgr';
10	CMO1 CMO3	CREATE MULTILEVEL OBJECT DaVinciCode IN DIMENSION product_dim AT LEVEL model UNDER Book SET price = 25, author = 'Brown'; CREATE MULTILEVEL OBJECT Illuminati IN DIMENSION product_dim AT LEVEL model UNDER Book SET price = 35, author = 'Brown'; CREATE MULTILEVEL OBJECT Sakrileg IN DIMENSION product_dim AT LEVEL model UNDER Book SET price = 45, author = 'Brown';

11	CMO1	CREATE MULTILEVEL OBJECT HarryPotter1 IN DIMENSION product_dim AT LEVEL model UNDER Book; CREATE MULTILEVEL OBJECT HarryPotter2 IN DIMENSION product_dim AT LEVEL model UNDER Book; CREATE MULTILEVEL OBJECT HarryPotter3 IN DIMENSION product_dim AT LEVEL model UNDER Book;
12	UMO3	UPDATE MULTILEVEL OBJECT HarryPotter1 IN DIMENSION product_dim SET price = 30, author = 'Rowling'; UPDATE MULTILEVEL OBJECT HarryPotter2 IN DIMENSION product_dim SET price = 35, author = 'Rowling'; UPDATE MULTILEVEL OBJECT HarryPotter3 IN DIMENSION product_dim SET price = 40, author = 'Rowling';
13	CMO1 CMO3	CREATE MULTILEVEL OBJECT Porsche911 IN DIMENSION product_dim AT LEVEL brand UNDER Car; CREATE MULTILEVEL OBJECT Porsche911Carrera IN DIMENSION product_dim AT LEVEL model UNDER Porsche911 SET price = 55000, maxSpeed = 320; CREATE MULTILEVEL OBJECT Porsche911GT3 IN DIMENSION product_dim AT LEVEL model UNDER Porsche911 SET price = 65000, maxSpeed = 350;
14	CMO2a	CREATE MULTILEVEL OBJECT Time IN DIMENSION time_dim AT LEVEL top HIERARCHY (LEVEL top CHILD OF NULL, LEVEL year CHILD OF top, LEVEL month CHILD OF year);
15	CMO1	CREATE MULTILEVEL OBJECT Year2012 IN DIMENSION time_dim AT LEVEL year UNDER Time; CREATE MULTILEVEL OBJECT JAN2012 IN DIMENSION time_dim AT LEVEL month UNDER Year2012; CREATE MULTILEVEL OBJECT FEB2012 IN DIMENSION time_dim AT LEVEL month UNDER Year2012;

16	CMO2c CMO4	CREATE MULTILEVEL OBJECT Location IN DIMENSION location_dim AT LEVEL top HIERARCHY (LEVEL top CHILD OF NULL, LEVEL country CHILD OF top, LEVEL region CHILD OF top, LEVEL city CHILD OF (country, region) (inhabitants NUMBER));
17	CMO1 CMO2a	CREATE MULTILEVEL OBJECT Austria IN DIMENSION location_dim AT LEVEL country UNDER Location; CREATE MULTILEVEL OBJECT Switzerland IN DIMENSION location_dim AT LEVEL country UNDER Location HIERARCHY (LEVEL country CHILD OF top, LEVEL kanton CHILD OF country, LEVEL city CHILD OF kanton);
18	CMO1 CMO3	CREATE MULTILEVEL OBJECT GermanSpeakingCountries IN DIMENSION location_dim AT LEVEL region UNDER Location; CREATE MULTILEVEL OBJECT Vaud IN DIMENSION location_dim AT LEVEL kanton UNDER Switzerland; CREATE MULTILEVEL OBJECT Lausanne IN DIMENSION location_dim AT LEVEL city UNDER Vaud, GermanSpeakingCountries SET inhabitants = 126000; CREATE MULTILEVEL OBJECT Salzburg IN DIMENSION location_dim AT LEVEL city UNDER Austria, GermanSpeakingCountries SET inhabitants = 150000;
19	CMC	CREATE MULTILEVEL CUBE sales_cube OF DIMENSIONS Product IN DIMENSION product_dim, Time IN DIMENSION time_dim, Location IN DIMENSION location_dim;
20	CMF1b CMF2a CMF2b CMF3b	CREATE MULTILEVEL FACT BETWEEN OBJECTS (Product, Time, Location) IN CUBE sales_cube HIERARCHY (CONNECTION LEVEL <model, month, city> (revenue NUMBER, qtysold NUMBER)) SET revenue.function = SUM, revenue.unit = Euro IN DIMENSION quantity_dim DEFAULT, qtysold.function = SUM;

21	CMF3a	CREATE MULTILEVEL FACT BETWEEN OBJECTS (DaVinciCode, JAN2012, Salzburg) IN CUBE sales_cube SET revenue = 100000, qtysold = 4000;
22	BCMF1 BUMF	CREATE MULTILEVEL FACT BETWEEN OBJECTS (Illuminati, JAN2012, Salzburg) IN CUBE sales_cube SET revenue = 100000, qtysold = 2857; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Sakrileg, JAN2012, Salzburg) IN CUBE sales_cube; UPDATE MULTILEVEL FACT BETWEEN OBJECTS (Sakrileg, JAN2012, Salzburg) IN CUBE sales_cube SET revenue = 100000, qtysold = 2500;
23	CMF1a	CREATE MULTILEVEL FACT BETWEEN OBJECTS (JAN2012 IN DIMENSION time_dim, Salzburg IN DIMENSION location_dim, HarryPotter1 IN DIMENSION product_dim) IN CUBE sales_cube;
24	CMF1b	REATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter2, JAN2012, Salzburg) IN CUBE sales_cube;
25	CMF1b UMF1	CREATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter3, JAN2012, Salzburg) IN CUBE sales_cube; UPDATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter1, JAN2012, Salzburg) IN CUBE sales_cube SET revenue = 200000, qtysold = 6666; UPDATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter2, JAN2012, Salzburg) IN CUBE sales_cube SET revenue = 300000, qtysold = 8571; UPDATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter3, JAN2012, Salzburg) IN CUBE sales_cube SET revenue = 400000, qtysold = 10000;

26	BCMF1 BUMF	CREATE MULTILEVEL FACT BETWEEN OBJECTS (DaVinciCode, FEB2012, Salzburg) IN CUBE sales_cube SET revenue = 50000, qtysold = 500; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Illuminati, FEB2012, Salzburg) IN CUBE sales_cube SET revenue = 120000, qtysold = 1200; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Sakrileg, FEB2012, Salzburg) IN CUBE sales_cube SET revenue = 70000, qtysold = 700; CREATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter1, FEB2012, Salzburg) IN CUBE sales_cube SET revenue = 150000, qtysold = 1500; CREATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter2, FEB2012, Salzburg) IN CUBE sales_cube SET revenue = 300000, qtysold = 3000; CREATE MULTILEVEL FACT BETWEEN OBJECTS (HarryPotter3, FEB2012, Salzburg) IN CUBE sales_cube SET revenue = 400000, qtysold = 4000; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Porsche911Carrera, JAN2012, Salzburg) IN CUBE sales_cube SET revenue = 3000000, qtysold = 3000; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Porsche911GT3, JAN2012, Salzburg) IN CUBE sales_cube SET revenue = 4000000, qtysold = 4000; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Porsche911Carrera, FEB2012, Salzburg) IN CUBE sales_cube SET revenue = 5000000, qtysold = 5000; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Porsche911GT3, FEB2012, Salzburg) IN CUBE sales_cube SET revenue = 4500000, qtysold = 4500;
27	CMC	CREATE MULTILEVEL CUBE currency_cube OF DIMENSIONS Product IN DIMENSION quantity_dim, Time IN DIMENSION time_dim;
28	CMF1b CMF2a CMF2b CMF3b	CREATE MULTILEVEL FACT BETWEEN OBJECTS (Currency, Time) IN CUBE currency_cube HIERARCHY (CONNECTION LEVEL <model, month, city> (Euro NUMBER, Dollar NUMBER)) SET Euro.function = AVG DEFAULT, Dollar.function = AVG DEFAULT;

29	BCMF1 BUMF	CREATE MULTILEVEL FACT BETWEEN OBJECTS (Euro, JAN2012) IN CUBE currency_cube SET Euro = 1, Dollar = 1.40; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Euro, FEB2012) IN CUBE currency_cube SET Euro = 1, Dollar = 1.45; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Dollar, JAN2012) IN CUBE currency_cube SET Euro = 0,6, Dollar = 1; CREATE MULTILEVEL FACT BETWEEN OBJECTS (Dollar, FEB2012) IN CUBE currency_cube SET Euro = 0,55, Dollar = 1;
30	QS1	SELECT * FROM ROLLUP[product_dim.category, time_dim.year, location_dim.country](sales_cube);
31	QS2	SELECT * FROM ROLLUP[product_dim.model, time_dim.year, location_dim.country](sales_cube), ROLLUP[Product_dim.category, Time_dim.year, Location_dim.country](sales_cube);
32	QS3	SELECT * FROM ROLLUP[product_dim.category, time_dim.year, location_dim.country] (SELECT MULTILEVEL * FROM sales_cube);
33	QS4	SELECT * FROM ROLLUP[product_dim.category, time_dim.year, location_dim.country] (SELECT MULTILEVEL revenue FROM sales_cube);
34	QS5	SELECT * FROM ROLLUP[product_dim.model, time_dim.month, location_dim.city] (revenue FROM DICE[Book, Year2012, Salzburg] (sales_cube));
35	QS6	SELECT * FROM ROLLUP[product_dim.category, time_dim.year, location_dim.country] (SELECT MULTILEVEL revenue FROM sales_cube s WHERE s.product_dim.model.price < 35);
36	QS7	SELECT * FROM ROLLUP[product_dim.category, time_dim.year, location_dim.country] (SELECT MULTILEVEL revenue FROM sales_cube s WHERE s.product_dim.model.price < 35 AND s.product_dim.model.author = 'Rowling');

37	QS8	SELECT * FROM ROLLUP[product_dim.category, time_dim.year, location_dim.city] (SELECT MULTILEVEL revenue FROM currency_cube c) CONVERT MEASURE revenue TO Dollar in quantity_dim USING currency_cube;

C. Manual

The chapter Manual addresses two subjects. First the installation of the SQL($\mathcal{M}$) interpreter as a Java stored procedure is explained. This is done via a step by step execution list. Then follows the presentation of the SQL($\mathcal{M}$) editor, a small graphical user interface, which uses the interpreter API to send queries and displays their results. It got developed as an example of how to use the interpreter API and can be used as a testing tool.

Installation

The SQL($\mathcal{M}$) interpreter is designed to run as a Java stored procedure in an Oracle 11g database. Before the interpreter can be used, the extension package for hetero-homogeneous dimension hierarchies and cubes has to be installed first. For information on the installation process of the package, please look at [HH-DW].

To be able to install and use the SQL($\mathcal{M}$) interpreter, an oracle user must possess certain database privileges:

- CONNECT and CREATE SESSION privileges for establishing a connection to the database.
- CREATE PROCEDURE and CREATE TABLE privileges to load into your schema.
- SELECT TABLE and INSERT TABLE privileges for data access.
- EXECUTE privilege for procedures and functions. This can be done two ways. One is to grant the EXECUTE ANY PROCEDURE privilege, the other manually granting EXECUTE privileges for all packages created by the extension or interpreter.

When all the needed privileges are granted, the installation of the SQL($\mathcal{M}$) interpreter can be carried out. The SQL($\mathcal{M}$) interpreter files are available is a .jar file. First these files need to be loaded into the database, and then be made accessible. To load Java files into the oracle database, the *loadjava* functionality of the Oracle database is used. The loading command is entered at the command prompt. The following example shows how a *loadjava* command looks like:

```
loadjava -user name/password -v -r -S file_path\Interpreter.jar
```

At the beginning stands the *loadjava* key word. Then -user (-u also works) follows, where user name and corresponding password is declared. After that some optional options follow. At the end the to be loaded jar file is declared. Some of the optional options are:

- verbose (-v): Enables the output of progress messages during the load phase.
- resolve (-r): The loaded Java files are getting compiled and external links in them are resolved after load up. If not enabled this is not done until run-time.

- schema (-S): Here the database schema can be specified for the Java files. If the option is not stated, then by default the logon schema is applied.

After the SQL($\mathcal{M}$) interpreter is loaded into the database, the Java files need to be published. In this case this is done through a function. For creating the function and the corresponding package, the *Interpreter.sql* file is used for. The file contains the call specifications for the callable Java method *sqlm_query* and the creation of the package *refcursor* for returning the query result.

SQL($\mathcal{M}$) editor

The SQL($\mathcal{M}$) editor is a grapical user interface, using the SQL($\mathcal{M}$) interpreter API to send SQL($\mathcal{M}$) queries and displaying the receiving result. For useing this tool, JavaFX is needed. JavaFX is available since Java SE 7 Update 6. When using only Java JRE the JavaFX package must be installed additionally. For further information about JavaFX visit [jfx].

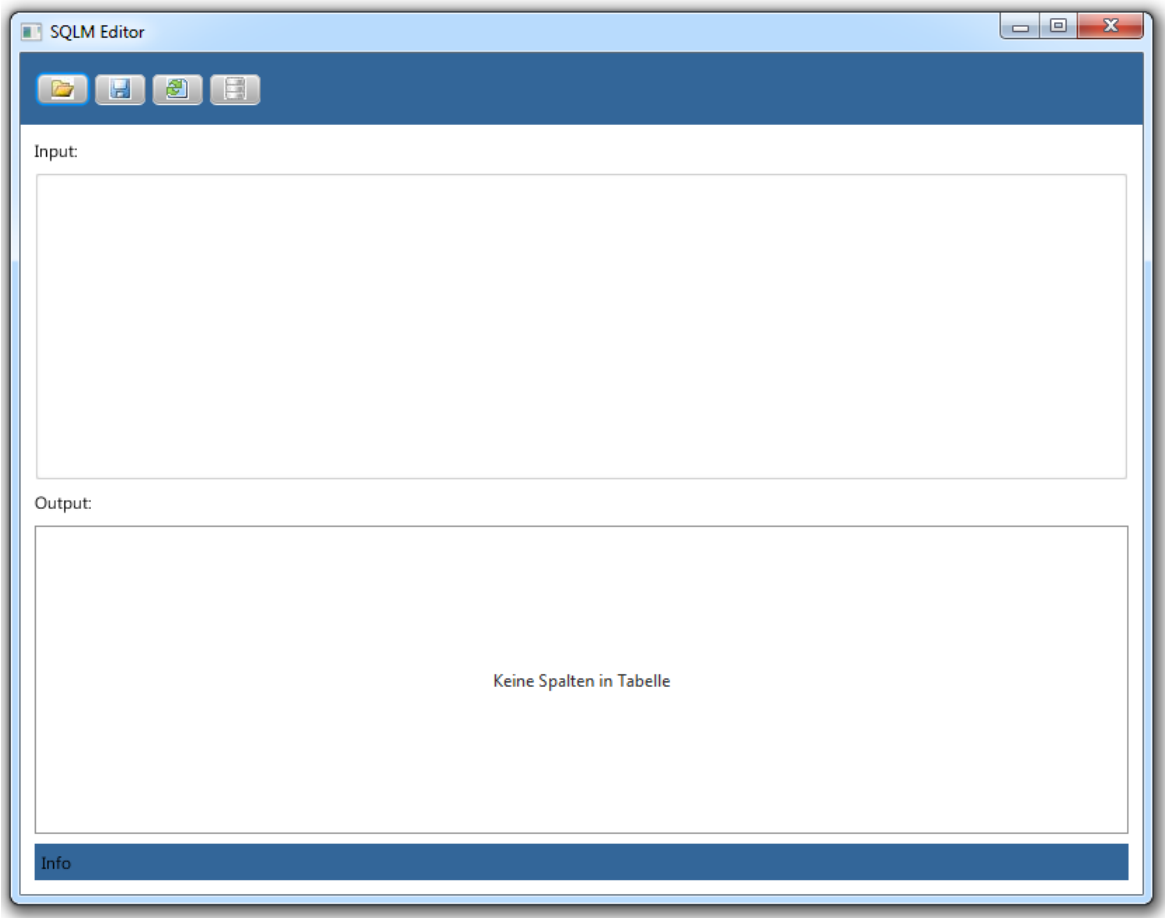


Figure 3.1.: SQL($\mathcal{M}$) editor main screen

Figure 3.1 shows how the SQL($\mathcal{M}$) editor looks like. The tool consists of four buttons for actions, an input panel, an output table and an info bar. Through the four buttons these functionalities of the editor can be accessed:

1. Load a text file into the input field.

2. Save the text inside the input field into a text file.

3. Send a query to the SQL($\mathcal{M}$) interpreter.

4. Define database and user properties.

Before queries can be send via the SQL($\mathcal{M}$) editor, the database and user properties (button four) must be declared. In Figure 3.2 the database properties menu is shown. There all the mandatory data can be defined. With clicking on submit, the information is stored inside a file located at the *My Documents* folder. The information inside this file is automatically loaded, when the editor is started.

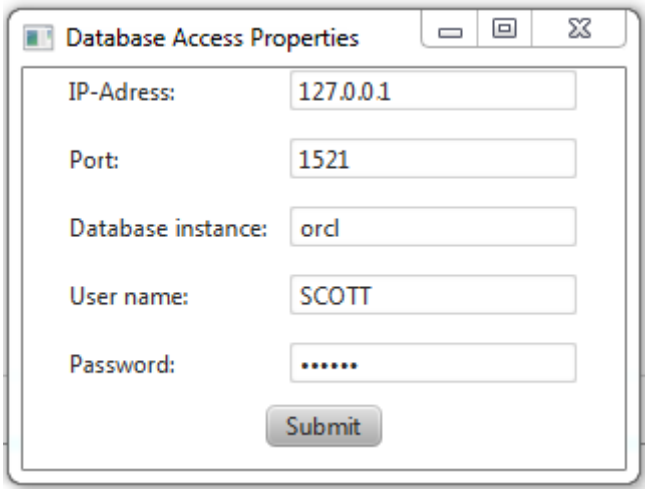


Figure 3.2.: Setting database properties in SQL($\mathcal{M}$) editor

The info bar at the bottom displays status information for the different functionalities of the editor. It shows if an action was successfully executed or states error information when a problem occurred.

Bibliography

[ALSU08] Alfred A. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compiler: Prinzipien, Techniken und Werkzeuge. Pearson Studium, second edition, 2008.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. ACM Sigmod record, 1997.

[Cod93] E.F. Codd, S.B. Codd and C.T. Salley. Providing OLAP to User-Analysts. E.F. Codd Associates, 1993.

[EN07] R. Elmasri, S.B. Navathe. Fundamentals of Database Systems. fifth edition, Pearson, 2007

[FAR11] Kiumars Farkisch. Data-Warehouse-Systeme kompakt - Aufbau, Architektur, Grundfunktionen. Springer Verlag, 2011.

[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.

[Inm02] W.H. Inmon, Building the Data Warehouse, fourth edition, New York: John Wiley & Sons, 2005.

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

[jav] Java Developers Perspective on Oracle Database 11g, An Oracle White Paper, September 2009

[jcc] Java Compiler Compiler - Documentation. Website. https://javacc.java.net/doc/

[jfx] JavaFX Frequently Asked Questions. Website. http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html

[MK00] Daniel L. Moody and Mark A.R. Kortink. From Enterprise Models to Dimensional Models: A Methodology for Data Warehouse and Data Mart Design. Proceedings of the International Workshop on Design and Management of Data Warehouses. June 2000.

[NGS09] Bernd Neumayr, Katharina Grün and Michael Schrefl. Multi-Level Domain Modeling with M-Objects and M-Relationships

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

[NST11] Bernd Neumayr, Michael Schrefl, and Bernhard Thalheim. Modeling Techniques for Multi-level Abstraction. Kaschek/Delcambre (Eds.): The Evolution of Conceptual Modeling, LNCS 6520, pp. 68-92, 2011. Springer-Verlag Berlin Heidelberg 2011.

[Odw07] Oracle Database Data Warehousing Guide 11g Release 1 (11.1). 2007.

[Ojd08] Oracle Database JDBC Developer's Guide and Reference 11g Release 1 (11.1). 2008.

[Oor08] Oracle Database Object-Relational Developer's Guide 11g Release 1 (11.1). 2008.

[Ora09] Oracle Database Java Developers Guide 11g Release 1 (11.1). 2009.

[PS99] J. Pokornỳ, P. Sokolowsky. Conceptual modelling perspective for datawarehouses. Wirtschaftsinformatik Proceedings - Paper 35, 1999.

[Pun09] A. A. Puntambekar. Compiler Design. Technical Publications, first edition, 2009.

[Rot15] J. Roth. Online Analytical Processing mit hetero-homogenen Data Warehouses: Prototypische Implementierung und Benutzerstudie. Diploma thesis. 2015.

[Sch10] Christoph Schütz. Extending data warehouses with hetero-homogeneous dimension hierarchies and cubes - A proof-of-concept prototype in Oracle. Diploma thesis. 2010.

[SRN15] Christoph G. Schuetz, Joachim Roth, Bernd Neumayr, Michael Schrefl: An OLAP Client for Hetero-Homogeneous Data Warehouses, Proceedings of 19th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, 2015.

[Ter96] P.D. Terry. Compilers and Compiler Generators - an introduction with C++. Rhodes University, 1996.

[HH-DW] Christoph Schütz. Prototyp eines hetero-homogenen Data Warehouses, 2011, available at: http://www.dke.uni-linz.ac.at/research/projects/hh-dw.html

[Wir77] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? Communications of the ACM 20(11), S. 822-823, 1977

[Wir08] N. Wirth. Grundlagen und Techniken des Compilerbaus. Oldenbourg Verlag, second edition, 2008.