JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Extending Data Warehouses with Reasoning over Multi-dimensional Ontologies: A Proof-of-Concept Prototype using PL/SQL and OWL

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Masterstudium

Wirtschaftsinformatik

Eingereicht von:
Christoph Ellinger

Angefertigt am:
Institut für Wirtschaftsinformatik - Data & Knowledge Engineering

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

Mitwirkung:
Dr. Bernd Neumayr

Linz, Oktober 2014

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit 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 Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, Oktober 2014

Christoph Ellinger

Abstract

In this thesis a prototype for the reasoning over multi-dimensional ontologies (MDO) is presented. The contribution of this thesis is to proof the feasibility of the introduced ideas by providing a research prototype that can be extended in the future. With the help of multi-dimensional ontologies an OLAP cube can be enriched with business terms, also called concepts. These concepts are in an unordered state when defined in the MDO. Reasoning support over multi-dimensional ontologies simplifies the use of these concepts by arranging them in hierarchies. An abstract syntax is defined in UML and implemented in a database schema, the MDO DB, this syntax represents the abstract MDO language. To ease the task of working with this prototype also a concrete syntax is implemented in ANTLR to interact with the prototype without knowing its inner behavior, this ANTLR implementation represents the concrete MDO language. The prototype shows the mapping of concepts from their MDO definition to OWL, with the concepts defined in OWL. A reasoner is used to create the subsumption hierarchy of the concepts. The thesis also provides a mapping from MDO to SQL views. With these views a standard data warehouse is enriched with business terms and these terms can be used in further queries. The prototype itself is also part of the Semantic Cockpit research project and implements main parts of it.

Zusammenfassung

Diese Masterarbeit beschreibt die Implementierung eines Prototyps zum Ableiten von Wissen über multi-dimensionale Ontologien. Der Beitrag dieser Arbeit ist die Machbarkeit der in dieser Arbeit vorgestellten Konzepte und Ideen mithilfe eines einfach erweiterbaren Forschungsprototypen darzustellen. Mithilfe multi-dimensionaler Ontologien können OLAP Würfel mit Geschäftsbegriffen angereichert werden. Um die Arbeit mit Geschäftsbegriffen zu erleichtern werden diese in Subsumtionsbeziehungen angeordnet. Es wird die abstrakte Syntax der multi-dimensionalen Ontologien in Form von UML Diagrammen und dem dazugehörigem Datenbank Schema dargestellt. Ebenfalls erklärt wird die Umsetzung der konkreten Syntax mithilfe des ANTLR Frameworks. Der Prototyp zeigt wie Geschäftsbegriffe in OWL und SQL abgebildet werden können. Mithilfe von SQL Sichten wird ein Data Warehouse mit Geschäftsbegriffen angereichert, welche in Abfragen gegen das Data Warehouse benutzt werden können. Der Prototyp ist im Rahmen des Semantic Cockpit Projektes entstanden und zeigt die Implementierung einiger Hauptkomponenten dieses Projektes.

Contents

1. Introduction

Large sets of data, for example a companies sales data, economic data or the data of a health insurer, are collected for the purpose of extracting knowledge. There is a variety of different tools available for collecting and analyzing data, with the data warehouse being one of the most commonly used tools. [Inmon, 1996, p.50] describes the data warehouse as a tool for 'successful and efficient exploration of the world of data'.

Data warehouses arrange data into measures and dimensions according to the dimensional modeling approach [Kimball and Strehlo, 1995]. A measure is the target for analysis, typical measures are costs, sales or quantities. These measures are then analyzed from different dimensions that can be seen as different perspectives on the measure, typical dimension are time, place or product. The different dimensions are organized in hierarchies and have different levels which represent different granularities of a dimension[Chaudhuri and Dayal, 1997]. A place dimension that looks at sales could be organized in store, sales district and state level.

These multi-dimensional data warehouses are queried with online analytical processing (OLAP) tools. If now many analysts work together on the same data they likely want to reuse some business terms they use in their queries.

[Neumayr et al., 2013] discovered the need for easing the task of writing OLAP queries in a collaborative working environment. Their proposal is to implement business terms into OLAP cubes by using the web ontology language (OWL) and standard relational database software.

1.1. The Semantic Cockpit Project

This thesis originated from the Semantic Cockpit research project[1], where academia and industry joined together to work on the Semantic Cockpit, to support business analysts in comparative data analysis with the help of domain ontologies. The main idea for the SemCockpit was proposed by [Neumayr et al., 2011], a comprehensive description of the research project is provided by [Neuböck et al., 2014]. SemCockpit uses semantic web technologies for exploratory OLAP, an overview over this topic is given by [Abello et al., 2014].

This master thesis is a result of the SemCockpit project, the multi-dimensional ontology engine, highlighted in figure 1. The MDO engine handles a multi-dimensional ontology (MDO) which unambiguously defines business terms. The engine is responsible for managing the MDO by defining and changing new business terms. It interacts with a reasoner to arrange the business terms in subsumption hierarchies and communicates with an underlying data warehouse which contains all the entities represented by the MDO. Another approach for reasoning over a restricted form of multidimensional ontologies was already implemented using Datalog [Neumayr et al., 2012]. Because the

[1]The Semantic Cockpit project was supported by the Austrian Ministry of Transport, Innovation, and Technology in the program FIT-IT Semantic Systems and Services under grant FFG-829594 Semantic Cockpit: An Ontology-Driven, Interactive Business Intelligence Tool for Comparative Data Analysis.

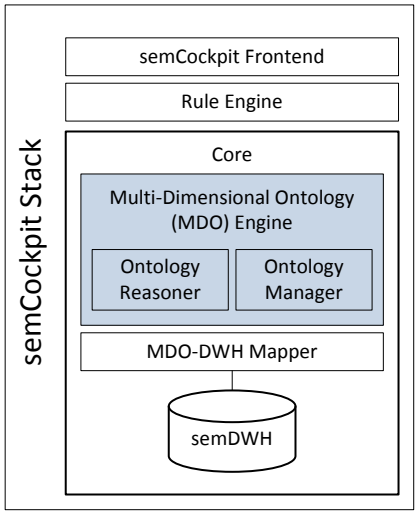

Figure 1: Abstract Semantic Cockpit architecture with parts highlighted that where implemented in this thesis

new approach extends the MDO language with disjunctions and complements of business terms the old approach is insufficient.

1.2. Running Example and Motivation

The idea behind this master thesis is the implementation of multi-dimensional reasoning support in data warehouses that was described in [Neumayr et al., 2013]. We want to show the feasibility of the described ideas and how we implemented them. The examples of this master thesis will be using simplified data of an Austrian health care provider, a project partner for the SemCockpit. The data warehouse consists of the dimensions Doctor, Insurant, Time and Drug and the structure of the data warehouse can be seen in figure 2. Every dimension consists of levels, for example Insurant consists of the levels insurant, district and province. Every level references entities of exactly one entity class. An entity class can be referenced by multiple entity classes, for example e_district is referenced by the district level of the Insurant dimension and the Doctor dimension. The dimension levels are organized in a roll-up hierarchy.

Data warehouse structures are often represented by relational views. In our example illustrated by figure 2 the following view structure is assumed: a fact view(drugPrescription) which stores the costs and quantities of prescribed drugs. The costs and quantity are stored at the most fine-grained level, additional measure views costs and granularity store the measure value on every possible granularity level. Additionally another measure view DrugCostsPerInsurant is defined which defines the drug costs per insurant independent of the Insurant dimension or the Drug dimension. The costs are stored for every granularity of the Doctor and Time dimension. The dimension are also views which reference entities at the levels of the dimension. The roll-up hierarchies are represented by roll-up views which define for every entity their superordinate entity. They define not

Relational Views:

drugPrescription(quanity, costs, insurant, drug, time, actDoc, leadDoc)
costs (costs, insurant, drug, time, actDoc, leadDoc)
quantity (costs, insurant, drug, time, actDoc, leadDoc)
DrugCostsPerInsurant(costs, time, actDoc, leadDoc)
Doctor(doctor, doctor_lvl)
Doctor_rollup (doctor, doctor_sup)
Insurant(insurant, insurant_lvl)
Insurant_rollup(insurant, insurant_sup)
Drug (drug, drug_lvl)
Drug_rollup (drug, drug_sup)
Time (time, time_lvl)
Time_rollup(time, time_sup)
e_district(district, inhabitants,sqkm,inhPerSqkm)
e_insurant(insurant, age, income)
e_doctor(doctor, age)
e_drug(drug, price)

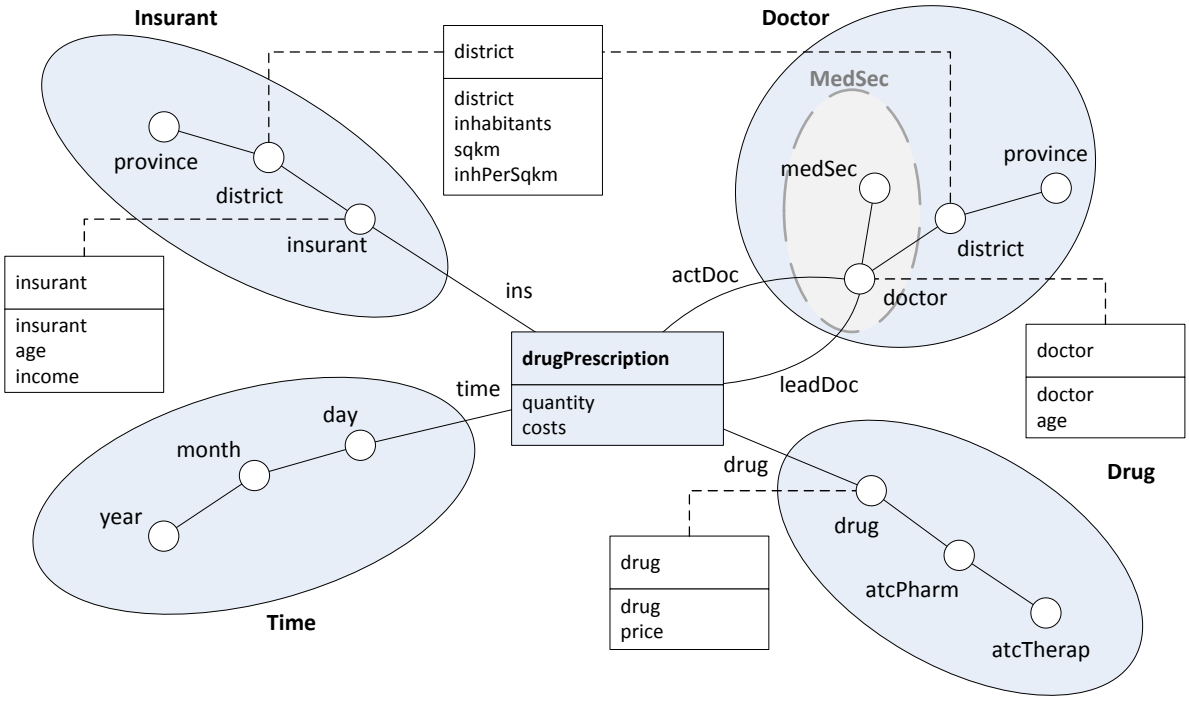

Figure 2: Data Warehouse Structure [Steiner, 2014, p.12] and Relational View Structure of the Data Warehouse

only direct but the indirect and reflexive roll-up relation between entities. Entity classes are represented by entity views and consist of the entities and their attribute values.

[Neumayr et al., 2013] proposed to represent the multi-dimensional form of a data warehouse with their multi-dimensional ontology. The MDO allows us to explicitly define business terms that are otherwise hidden in the data structure of a data warehouse. To show what we mean with hidden we look at an example query in listing 1, the query is written in SQL and does not support concepts:

```
SELECT * FROM DrugCostsPerInsurant
        WHERE actDoctor IN (
                SELECT doctor FROM Doctor_rollup
                WHERE Doctor_sup IN (
                SELECT district FROM e_district WHERE inhabitants>1000000
                    ) )
                AND doctor IN (
                SELECT doctor FROM Doctor_rollup
                WHERE Doctor_sup IN (
                SELECT e_province FROM e_province WHERE inhabitants
                   <20000000 ) )
                AND doctor IN (
                SELECT doctor FROM Doctor WHERE Doctor lvl = 'district' )
                    )
        AND time = '2010'
```

Listing 1: OLAP Query without concepts

[Neumayr et al., 2013] found some issues identified with this example. First we see that the query itself looks rather complex and the meaning of the query is not easy to grasp at the first look. To make further reading more easy here the explanation: This query gets the costs per insurant, for insurants, where their doctor for the treatment works in a big district with more than one million inhabitans and the province in which he practices is a small province with less then twenty million inhabitants. The costs of the insurants shall be aggregated to the district level and the year of the costs should be 2010. The costs are aggregated over all drugs. A second problem we can see is that the business terms the analyst uses here (big district, small province) are hard coded into the query. This can lead to mistakes as the business analyst writing the query has to use the same numbers in every query to make the results comparable. Also it makes the analysis very hard to change, if in some time in the future the definition for a big district changes for whatever reason, the business analyst has to change every number in all the queries he wrote.

For this reason [Neumayr et al., 2013] introduced so called concepts. These concepts are the business terms we introduced before, e.g. big district or small province. Our concepts are defined for different components of the data warehouse, we distinguish between three main concept types: entity concepts, dimensional concepts and multi-dimensional concepts. Entity concepts are defined for concrete entities, for example a district or a doctor and are not bound to any dimensions yet. For example the entity concept 'big district' can later be used for the insurant dimension or the doctor dimension, as they share the same entities. Dimensional concepts are bound to the nodes of a specific dimension, for example we could bind the big district entity concept to the doctor dimension. Multi-dimensional concepts are concepts that span over a certain number of dimensions and can be applied to points of the data warehouse. Multi-dimensional concepts do not refer to dimensions directly but certain dimension roles that are applied on a dimension, for instance we defined two dimension roles for the doctor dimension. One for the acting doctor and another role for the lead doctor, therefore the same dimension can be used as often as necessary to get the desired data representation. The whole arrangement of

these concepts and the representation of the data warehouse is our MDO. The goal of this implementation is to use our concepts, defined in the MDO, in our queries. How such a query could look like is shown in listing 2.

```
SELECT * FROM DrugCostsPerInsurant
        NATURAL JOIN actDocInBigDistrictAndSmallProvince
        WHERE doctor IN (
        SELECT Doctor FROM Doctor WHERE Doctor lvl = 'district' )
        AND time = '2010'
```

Listing 2: Query using MDO concepts

The new query incorporates the business terms now with the use of concepts. We created the multi-dimensional concept actDocInBigDistrictAndSmallProvince, that consists of our two business terms big district and small province. With this approach we solved the problems discussed in the section above, the query is now easier to read, the terms are unambiguously defined and we can reuse the terms in other queries.

Another problem occurs if the user starts defining more concepts. With every new concept it gets harder for the user to identify the relationships between the concepts or to see which concepts are a generalization of other concepts. For this reason we implemented the OWL-based reasoner which arranges our defined concepts into subsumption hierarchies. The OWL-based reasoner is described in section 4. An organization of the example concepts used in this thesis is shown in figure 3.

Throughout the thesis we will look at a running example to show how concepts can be defined and how they are mapped to ANTLR, OWL and the data warehouse. We use the anonymized and simplified data of a health care provider as test data and are therefore in a health care setting. The example are all built in a virtual environment[2] where all basic components of the data warehouse and MDO-DB already exist, therefore we do not have to define our own entity classes, dimensions etc. The definition of the running example is in section 2.3 under the respective concept types. Over the course of this master's thesis we will issue a couple of commands to the user interface, to keep every section compact we will only show a small parts of the commands issued. The whole list of commands issued for creating the test concepts can be found in the Appendix A.

The next section shows a general architecture of our prototype.

1.3. General Architecture

The basis underlying the whole implementation of this thesis is an Oracle database with two schemas: the semantic data warehouse (semDWH) and the multi-dimensional ontology database (MDO-DB). The semDWH contains all the data migrated from a conventional data warehouse. In addition to the classic data warehouse data it is enriched by concept views i.e representations of concept definitions as SQL views. The semDWH structure is described in section 5.

The MDO-DB contains the definitions of our concepts, the mappings of these concepts to OWL and the mappings for the semDWH. It also contains the MDO representation

[2]The virtual machine running the environment can be found on the DVD enclosed to this thesis

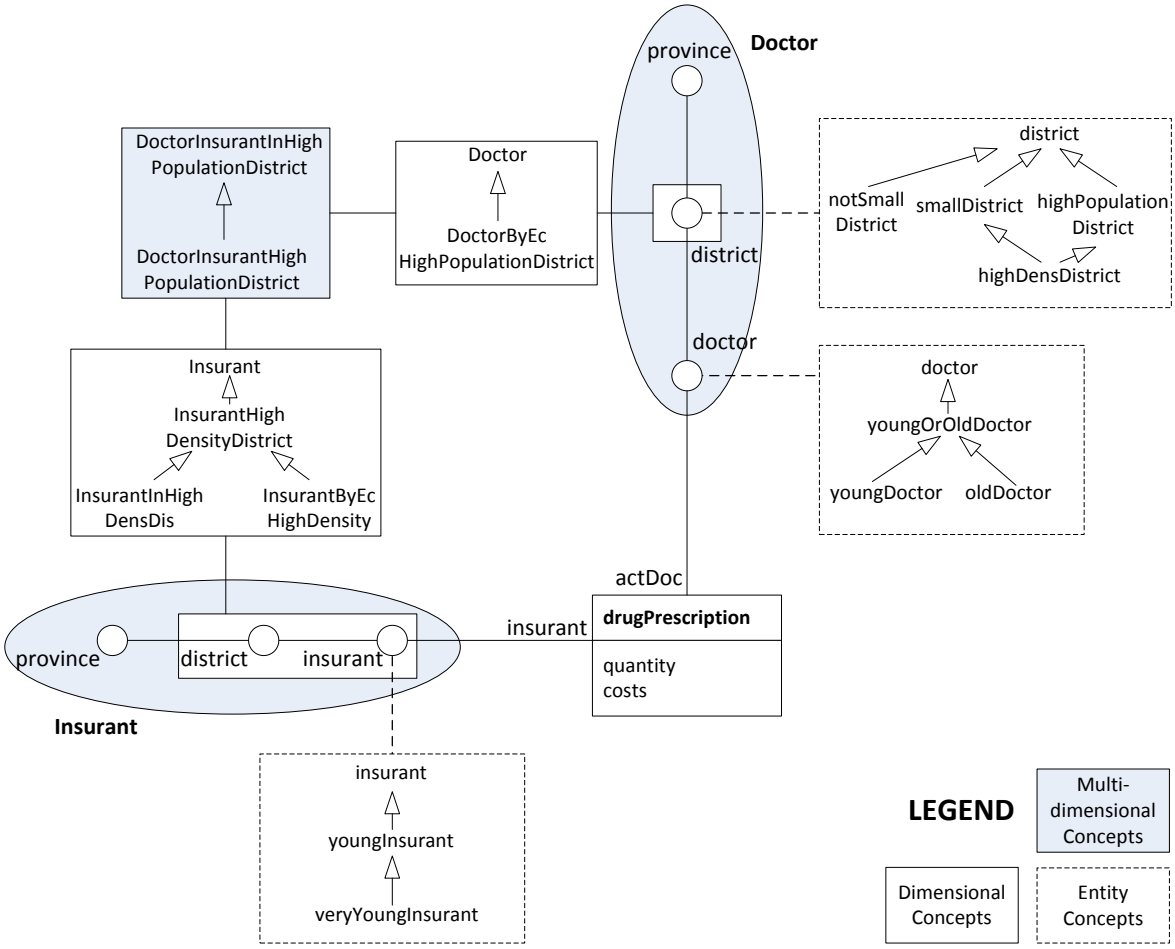

Figure 3: Concept organization of the defined example concepts

of our data warehouse. Additionally the MDO-DB interacts with our user interface, the semDWH and the OWL-based reasoner. An overview over the semCockpit architecture can be found in figure 4.

To interact with the prototype we assume a correctly initialized database where all the data from a standard data warehouse is transferred into the semantic data warehouse and where the structure of the data warehouse is also represented in the MDO-DB. Via the ANTLR-based user interface, the user can create new concepts for the users specific needs. After creating a few concepts, the user can start the reasoning process via the user interface. The MDO-DB then initializes the reasoner via user input and the reasoner organizes the concepts in subsumption hierarchies. If the user wants to see which instances are members of the newly created concepts the user may start the mapping process to the semDWH via the user interface. The MDO-DB then creates the needed views in the semDWH an the user can inspect these newly created concepts inside the semantic data warehouse.

1.4. Regarding Performance Tests

Although performance was not the main issue when creating this protoype we still wanted to see how our program behaved with a rising number of concepts. We have no single section for performance tests but every subsection has its own performance tests.

We conducted our tests with the following setup. As host machine we used a laptop, running Windows 8.1 (x64) with a 2,4GHz 4-core Intel i7 processor, 8 gigabyte of ram and a conventional HDD. The Java and reasoning applications run on this laptop. For the databases containing the MDO-DB and the SemDWH we installed a virtual environment on the laptop. The environment was a VirtualBox VM running Linux 6.4 (x64) with 2 Gigabyte RAM 1 CPU and hardware virtualization enabled.

For our tests we created a number of concepts (from 50 up to 175) and inserted them via our Java application, we then measured how long it took the prototype to complete the desired operation with the different amount of concepts inserted. The measurement of the execution time was carried out by the Java prototype. We measured the time the implementation needs to: write the concepts into the MDO-DB and map them (ANTLR-based Parser), derive the subsumption hierarchy (OWL-based reasoner) and create the concept in the SemDWH (MDO-DWH Mapper). The measurements are split into the different concept types, entity concepts, dimensional concepts and multi-dimensional concepts, to see if we have a performance difference between these concepts.

For every test case we created at least five different sets of concepts; the averaged runtime results are depicted in the respective sections.

1.5. Thesis Structure

The main building blocks of the architecture shown in figure 4 are also the main building blocks for this thesis.

In section 2 we will show how our MDO-DB is structured and how it works. We will show the SQL statements needed to create the concepts and introduce our running

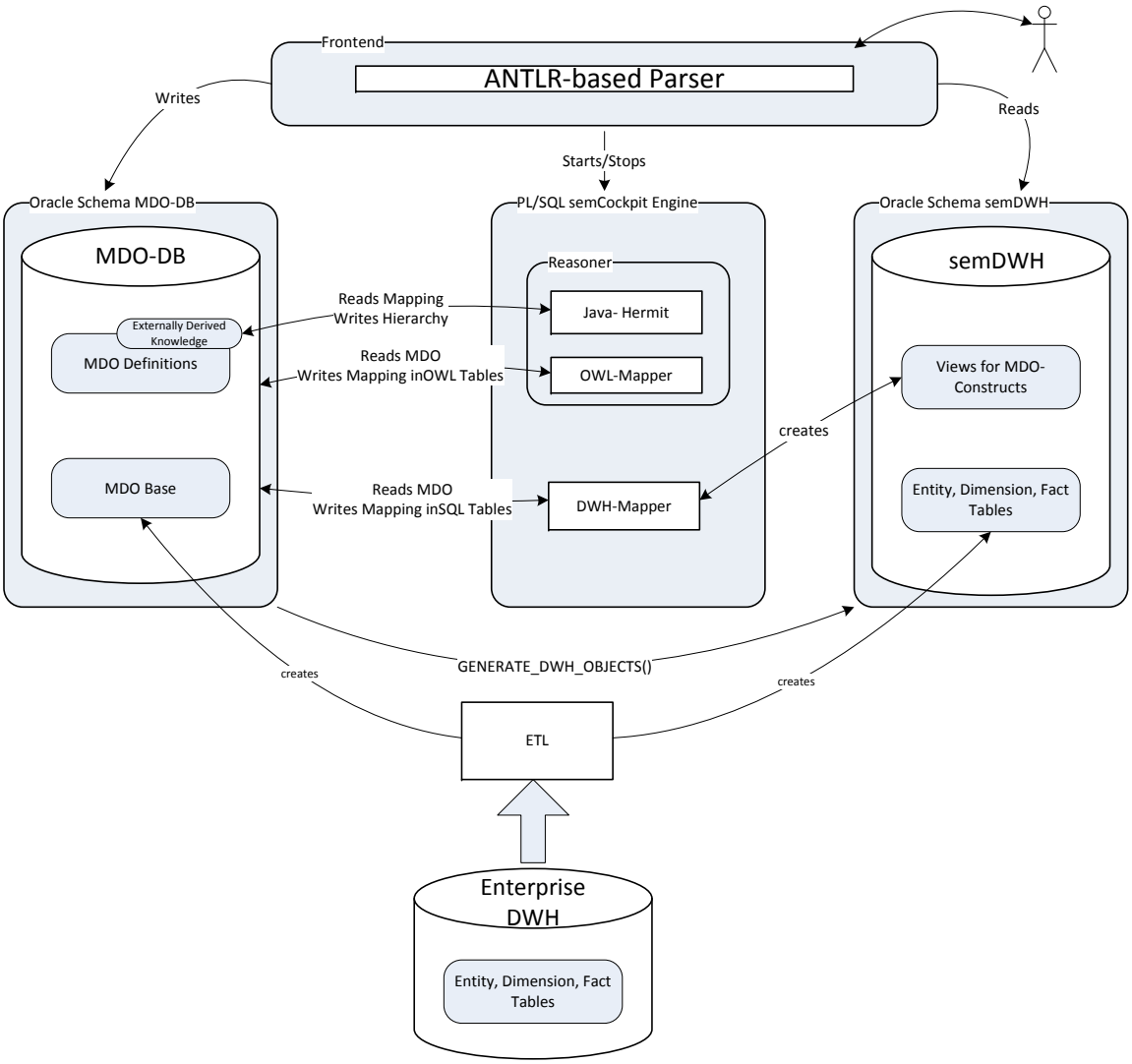

Figure 4: Architectural Overview

example. The MDO-DB acts as the abstract MDO Syntax and is specified using UML class diagrams and the corresponding DDL statements to create the MDO-DB.

Section 3 shows how we implemented the concrete MDO Syntax. For this purpose we used ANTLR to create a parser for the MDO language, this parser supports the user in generating MDO concept definition DML statements. We will show how we defined the grammar for the parser and how the parser was implemented in Java.

Section 4 describes the mapping of MDO concepts to OWL in Description Logic and Manchester Syntax. The section also shows the implementation of the OWL mapping inside the MDO-DB and the reasoning process outside the MDO-DB with off-the-shelf reasoner support and the OWL-API.

Section 5 first describes the structure of the semantic data warehouse. Then the mapping of the concepts in the MDO-DB to their concept view representation in SQL is shown.

Section 6 contains the implementation of contextualized and contextspecific concepts. These concepts are a special kind of concepts. They where not described by [Neumayr et al., 2013] and only briefly mentioned in [Neuböck et al., 2014] and have therefore a special status inside the thesis. In the last section we will give a further outlook over the topic.

2. MDO-DB: SQL-based management of multidimensional ontologies

This section gives an overview of the MDO-DB, a relational database representing a multi-dimensional ontology (MDO). An MDO-DB contains all concept definitions and also an MDO representation of the underlying data warehouse. It acts as intermediary between the semantic data warehouse (semDWH) and the user interface. We will show the data structure of the MDO-DB and the sample data we will use to test the different parts of our implementation.

2.1. Overview

Figure 4 shows that the MDO-DB can be separated into two parts, the MDO Base and the MDO Definitions. The MDO Base is a representation of an OLAP Cube, its dimensions, hierarchies et cetera. and is described in section 2.2. MDO Definitions are all concept definitions, their mapping to OWL and their mapping to concept views in the semDWH. How these concepts are represented in the MDO-DB is shown in section 2.3. Additionally the MDO-DB contains all the stored procedures and triggers for creating and maintaining the OWL and semDWH Mappings. The specifics how the OWL and semDWH mapping is done in the MDO-DB and how the mapping procedures interact with the MDO-DB is shown in their respective sections(for OWL see section 4 and for the semDWH see section 5). The component responsible for creating the inserts into the MDO-DB is the ANTLR-based user interface which is described in section 3.

2.2. Conceptual representation of OLAP cubes

The MDO-Base is a representation of a data warehouse OLAP cube to a multi-dimensional ontology. The data model for the OLAP cube is taken from [Neumayr et al., 2013, p.3] and consists of five basic building blocks: entity classes, dimensions, dimension roles, dimension spaces and measures. An entity class has a name and a number of attributes, an attribute has a certain data type. Dimensions consist of several levels and the levels are organized per dimension in a roll up hierarchy. A roll up hierarchy has exactly one top level and exactly one bottom level. A level refers to exactly one entity class. A dimension space consists of multiple dimensions where a dimension can play different roles, so the dimension space is the cartesian product of a number of dimension roles. Measures are defined for dimension spaces.

An OLAP cube also contains different kinds of instances like entities, nodes and points. An entity is member of an entity class and has concrete data values for the attributes defined in the entity class. A node is member of a dimension at a specific level, and refers to a concrete entity. The entity referred to by the node must be from the entity class referred to by the level of the node. Nodes are organized in roll up hierarchies similar to level hierarchies. The interpretation of a node is that it is the role an entity plays in the specific dimension. An entity can only be referred to by one node per dimension.

Figure 5 defines the structure of the MDO Base parts of the MDO-DB in terms of a UML class diagram, listing 3 shows the DDL statements to create this schema. To keep the listings more compact we omit names of constraints in the listing.

A thing to notice when looking at the DDL statemens for creating the MDO-DB is that virtual columns are used as primary keys. In listing 3 line 8 the primary key is a virtual column defined as the attributes name, and the entity class the attribute refers to, separated by a dot. We did not want to use compound keys to make further referencing of such constructs easier. To accomplish this we could have used numbers as surrogate keys but we wanted that the key values carry a meaning, therefore we chose to implement them using virtual columns. To keep the implementation consistent virtual columns where used as primary keys on nearly all tables, even if they where not necessary, as for example in listing 3 line 2 for the definition of the entity class key. Virtual columns do not allow to reference only one column[3] therefore we concatenated the column we referenced with an empty string which can also be seen in listing 3 line 2. In the implementation the column name of a primary key always ends with 'ID' and is constructed using a virtual column.

Another thing to notice is that we used prefixes to separate the different parts of the MDO-DB. MDO Base constructs (both classes and instances) start with the prefix 'dw_', entity concepts with 'ec_', dimensional concepts with 'dc_' and multi-dimensional concepts with 'mdc_'.

The UML diagram in figure 5 is mostly implemented in a straightforward way. One difference between the previous explanations and the implementation is the roll up hierarchy between levels. In the implementation we do not save all transitive roll up dependencies between the levels but only the direct roll-up relation as shown in listing 3 line 38. To ensure that only levels of the same dimension roll-up to one another we built a check constraint that uses regular expressions, see line 48. The primary key of a level is defined as 'dimensionID' dot 'levelID', with the regular expression we get the part of the level ID before the dot, the dimension ID. If these dimension ids are equal the constraint is fulfilled.

Another difference is the existence of two different dimension role types. One dimension role type is defined on the whole dimension(line 83) the other dimension role type is defined only for a certain hierarchy of a dimension. This difference is needed when alternative roll up hierarchies on one dimensions are defined and the dimension role should only cover one hierarchy. We implemented the different types with a base class (line 75) that contains only the name of the dimension role and the type of dimension role, checked by a constraint.

The instance definitions in listing 4 do not differ much from the UML diagram. Like the roll-up relation between levels, only the direct roll-up relation between nodes is saved in the MDO-DB (listing 4 line 38).

[3]More information on virtual columns can be found at `http://www.oracle-base.com/articles/11g/virtual-columns-11gr1.php`

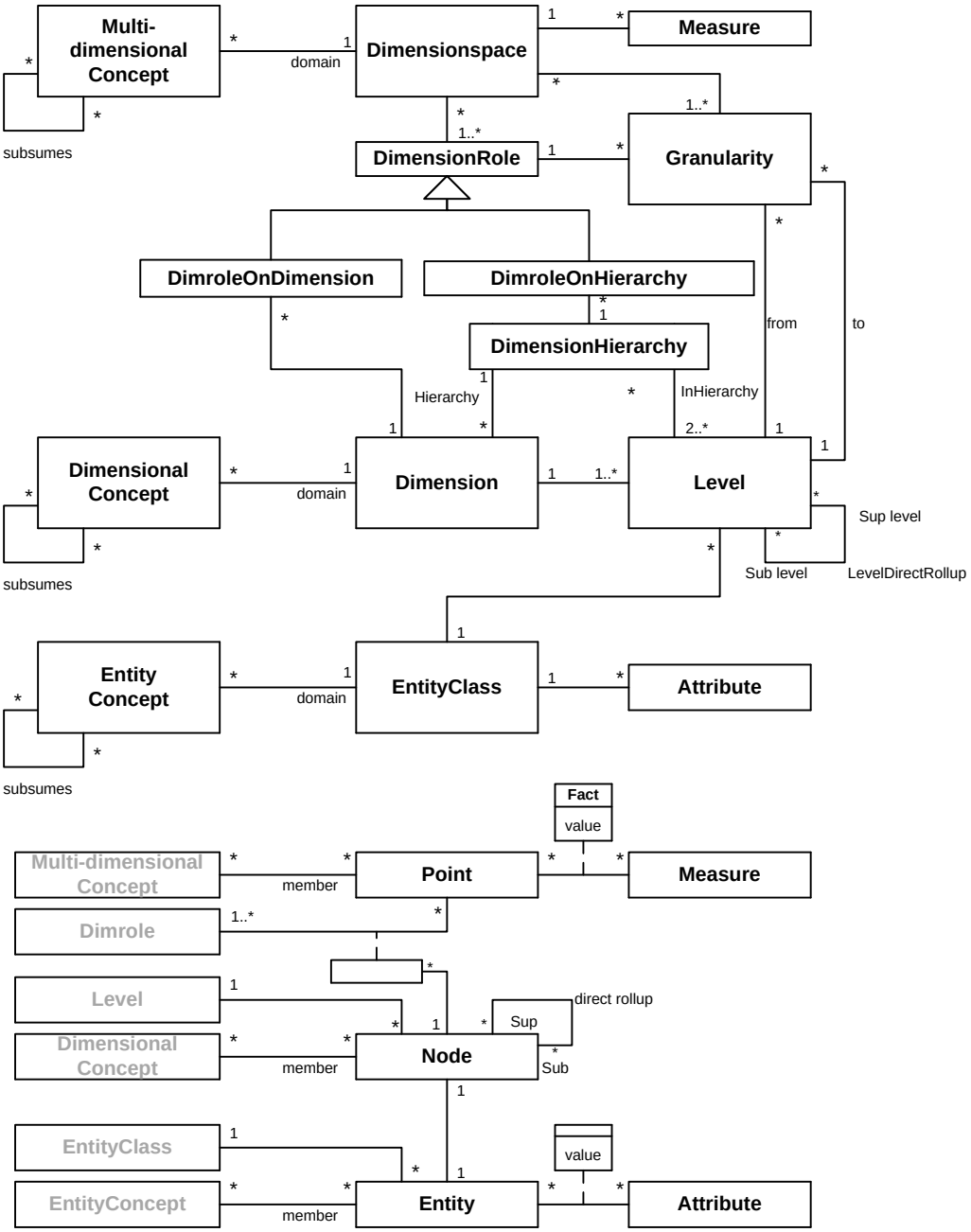

Figure 5: MDO Base Overview Schema perspective (top) and instance perspective (bottom)

```
CREATE TABLE dw_entityclass(
 entityclassID VARCHAR2(100) GENERATED ALWAYS AS (
      entityclassName ||'') VIRTUAL,
 entityclassName VARCHAR2(100) NOT NULL,

 CONSTRAINT PRIMARY KEY(entityclassID)
 );
CREATE TABLE dw_attribute (
  attributeID VARCHAR2(201) GENERATED ALWAYS AS (
       entityclassID || '.' || attributeName)
       VIRTUAL,
  entityclassID VARCHAR2(100) NOT NULL,
  attributeName VARCHAR2(100) NOT NULL,
  datatype VARCHAR2(100) NOT NULL,
  orderby_direction VARCHAR2(5),
  orderby_nr INTEGER,

  CONSTRAINT PRIMARY KEY (attributeID),
  CONSTRAINT FOREIGN KEY (entityclassID)
  REFERENCES dw_entityclass(entityclassID),
  CONSTRAINT datatype_in CHECK(datatype IN ('string
       ','integer','float','date')),
  CONSTRAINT direction_check CHECK (
       orderby_direction IN ('ASC','DESC'))
);
 CREATE TABLE dw_dimension(
 dimensionID VARCHAR2(100) GENERATED ALWAYS AS (
      dimensionName ||'') VIRTUAL,
 dimensionName VARCHAR2(100) NOT NULL,
 CONSTRAINT PRIMARY KEY(dimensionID)
 );
 CREATE TABLE dw_level(
  levelID varchar2(201) GENERATED ALWAYS AS (
       dimensionID||'.'||levelName),
  levelName varchar2(100) NOT NULL,
  dimensionID varchar2(100) NOT NULL,
  entityclassID varchar2(100) not null,

  CONSTRAINT FOREIGN KEY (dimensionID)
  REFERENCES dw_dimension(dimensionID),
  CONSTRAINT FOREIGN KEY(entityclassID)
  REFERENCES dw_entityclass(entityclassID),
  CONSTRAINT PRIMARY KEY(levelID))
 );
CREATE TABLE dw_level_directrollup(
  sub_levelID varchar2(201),
  sup_levelID varchar2(201),

CONSTRAINT PRIMARY KEY (sub_levelID,sup_levelID),
  CONSTRAINT FOREIGN KEY (sub_levelID)
  REFERENCES dw_level (levelID),
  CONSTRAINT(sup_levelID)
  REFERENCES dw_level (levelID),
  --Check if same dimension
  CONSTRAINT rollup_dimensioncheck CHECK (
       REGEXP_REPLACE(sub_levelID, '(\..*)')
  LIKE REGEXP_REPLACE(sup_levelID, '(\..*)'))
 );
CREATE TABLE dw_hierarchy(

  hierarchyID varchar2(201) GENERATED ALWAYS AS (
       dimensionID||'.'||hierarchyName),
  dimensionID VARCHAR2(100) not null,
  hierarchyName VARCHAR2(100) not null,

  CONSTRAINT PRIMARY KEY(hierarchyID),
  CONSTRAIN FOREIGN KEY (dimensionID)
  REFERENCES dw_dimension(dimensionID)
);
CREATE TABLE dw_inHierarchy(
  hierarchyID varchar2(201),
  levelID VARCHAR2(201),

  CONSTRAINT PRIMARY KEY(hierarchyID,levelID),
  CONSTRAINT FOREIGN KEY (hierarchyID)
  REFERENCES dw_hierarchy (hierarchyID),
  CONSTRAINT FOREIGN KEY (levelID)
  REFERENCES dw_level (levelID),
  --check if same dimension
  CONSTRAINT inhierarchy_dimensioncheck CHECK (
       REGEXP_REPLACE(hierarchyID, '(\..*)')
  LIKE REGEXP_REPLACE(levelID, '(\..*)'))
);

CREATE TABLE dw_dimrole (
  dimroleID VARCHAR2(100) GENERATED ALWAYS AS (
       dimroleName ||'') VIRTUAL,
  dimroleName VARCHAR2(100) NOT NULL,
  discriminator VARCHAR2(30) NOT NULL,

  CONSTRAINT PRIMARY KEY(dimroleID),
  CONSTRAINT dw_dimrole_disc CHECK(discriminator IN
       ('dimroleondimension','dimroleonhierarchy')
       )
);
CREATE TABLE dw_dimroleOnDimension (
  dimroleID VARCHAR2(100) NOT NULL,
  dimensionID VARCHAR2(100) NOT NULL,
  CONSTRAINT PRIMARY KEY(dimroleID),
  CONSTRAINT FOREIGN KEY(dimensionID)
  REFERENCES dw_dimension(dimensionID),
  CONSTRAINT FOREIGN KEY(dimroleID)
  REFERENCES dw_dimrole(dimroleID)
);
 CREATE TABLE dw_dimroleOnHierarchy(

  dimroleID VARCHAR2(100) NOT NULL,
  hierarchyID VARCHAR2(201) NOT NULL,
  hierarchyOnDimroleID VARCHAR2(100) NOT NULL,

  CONSTRAINT PRIMARY KEY(dimroleID),
  CONSTRAINT FOREIGN KEY(hierarchyOnDimroleID)
  REFERENCES dw_dimrole(dimroleID),
  CONSTRAINT FOREIGN KEY(hierarchyID)
  REFERENCES dw_hierarchy(hierarchyID),
  CONSTRAINT FOREIGN KEY(dimroleID)
  REFERENCES dw_dimrole(dimroleID),
  CONSTRAINT UNIQUE (hierarchyID,
       hierarchyOnDimroleID)
 );

CREATE TABLE dw_dimspace (
  dimspaceID VARCHAR2(100) GENERATED ALWAYS AS (
       dimspaceName ||'') VIRTUAL,
  dimspaceName VARCHAR2(100) NOT NULL,

  CONSTRAINT PRIMARY KEY(dimspaceID),
);

CREATE TABLE dw_Granularity(
  dimspaceID VARCHAR2(100),
  dimroleID VARCHAR2(100),
  from_levelID VARCHAR2(201) NOT NULL,
  to_levelID VARCHAR2(201) NOT NULL,

  CONSTRAINT FOREIGN KEY (dimroleID)
  REFERENCES dw_dimrole(dimroleID) ,
  CONSTRAINT FOREIGN KEY(from_levelID)
  REFERENCES dw_level(levelID),
  CONSTRAINT FOREIGN KEY(to_levelID)
  REFERENCES dw_level(levelID) ,
  CONSTRAINT FOREIGN KEY(dimspaceID)
  REFERENCES dw_Dimspace(dimspaceID),
  CONSTRAINT PRIMARY KEY(dimspaceID,dimroleID)
);
CREATE TABLE dw_measureInFactclass(

  fMeasureID VARCHAR2(201) GENERATED ALWAYS AS (
       factclassID || '.' || etlMeasureName )
       VIRTUAL,
  factclassID VARCHAR2(100) NOT NULL,
  --Name of the measure in the original data
       warehouse, loaded by the etl process
  etlMeasureName VARCHAR2(100) NOT NULL,
  dimspaceID VARCHAR2(100) NOT NULL,
  numericdatatype VARCHAR2(100) NOT NULL,

  CONSTRAINTS FOREIGN KEY(factclassID)
  REFERENCES dw_factclass(factclassID),
  CONSTRAINTS FOREIGN KEY(dimspaceID)
  REFERENCES dw_monogranulardimspace(dimspaceID)
       DEFERRABLE INITIALLY DEFERRED,
  CONSTRAINTS PRIMARY KEY (fMeasureID),
  CONSTRAINT dw_measureInFact_datatype
  CHECK(numericdatatype IN ('float','integer','
       double'))
);
```

Listing 3: DDL statements for creating the MDO Base classes

```
1
2  CREATE TABLE dw_entity(
3    entityID varchar2(201) GENERATED ALWAYS AS (
           entityclassID || '.' || entityName)
           VIRTUAL,
4    entityName varchar2(100) not null,
5    entityclassID varchar2(100) not null,
6
7    CONSTRAINT PRIMARY KEY(entityID),
8    CONSTRAINT FOREIGN KEY (entityclassID)
9    references dw_entityclass (entityclassID)
10 );
11
12 CREATE TABLE dw_attributevalue(
13   entityID varchar2(201) not null,
14   attributeID varchar2(201) not null,
15   attributevalue varchar2(100) not null,
16
17   CONSTRAINT PRIMARY KEY(entityID,attributeID),
18   CONSTRAINT FOREIGN KEY (entityID)
19   references dw_entity(entityID),
20   CONSTRAINT FOREIGN KEY (attributeID)
21   references dw_attribute(attributeID)
22 );
23 CREATE TABLE dw_node(
24   nodeID varchar2(302) GENERATED ALWAYS AS (
         dimensionID || '.' || entityID) VIRTUAL,
25   dimensionID varchar2(100) not null,
26   entityID varchar2(201) not null,
27   levelID varchar2(201) not null,
28
29   CONSTRAINT PRIMARY KEY(nodeID),
30   CONSTRAINT FOREIGN KEY(dimensionID)
31   REFERENCES dw_dimension(dimensionID),
32   CONSTRAINT FOREIGN KEY (levelID)
33   references dw_level(levelID),
34   CONSTRAINT FOREIGN KEY (entityID)
35   references dw_entity (entityID)
36 );
37
38 CREATE TABLE dw_node_directrollup(
39  sup_nodeID varchar2(302) not null,
40  sub_nodeID varchar2(302) not null,
41
42  CONSTRAINT PRIMARY KEY(sup_nodeID,sub_nodeID),
43  CONSTRAINT FOREIGN KEY (sup_nodeID)
44  REFERENCES dw_node(nodeID),
45  CONSTRAINT FOREIGN KEY (sub_nodeID)
46  REFERENCES dw_node(nodeID)
47 );
48 CREATE TABLE dw_point(
49   pointID varchar2(100) GENERATED ALWAYS AS (
         pointName || '') VIRTUAL,
50   pointName varchar2(100) not null,
51
52   CONSTRAINT PRIMARY KEY(pointID)
53 );
54 CREATE TABLE dw_point_dimrole(
55   pointID varchar2(100) not null,
56   dimroleID varchar2(100) not null,
57   nodeID varchar2(302) not null,
58
59   CONSTRAINT PRIMARY KEY(pointID,dimroleID),
60   CONSTRAINT FOREIGN KEY(dimroleID)
61   references dw_dimrole(dimroleID),
62   CONSTRAINT FOREIGN KEY(nodeID)
63   references dw_node(nodeID),
64   CONSTRAINT FOREIGN KEY(pointID)
65   references dw_point(pointID)
66 );
```

Listing 4: DDL statements for creating the MDO Base instances

Figure 6 shows the object diagram for a small part of the MDO Base. Listing 5 contains the DML statements needed to create the instances of this object diagram. In the MDO-DB in general, instances for entities, nodes and points are not stored, as they are saved and being processed by the semantic data warehouse. The only exception why a small number of instances is saved in the MDO-DB is so the instances can be used when they are needed in a concept definition. For the structure of our concepts in the MDO-DB see the next section 2.3.

The insertion of instances into the MDO-DB is straightforward. The only thing to keep in mind is that virtual columns are used, because of this, the column names cannot be omitted when inserting into the MDO-DB. As shown in listing 5 all insert statements use the column names.

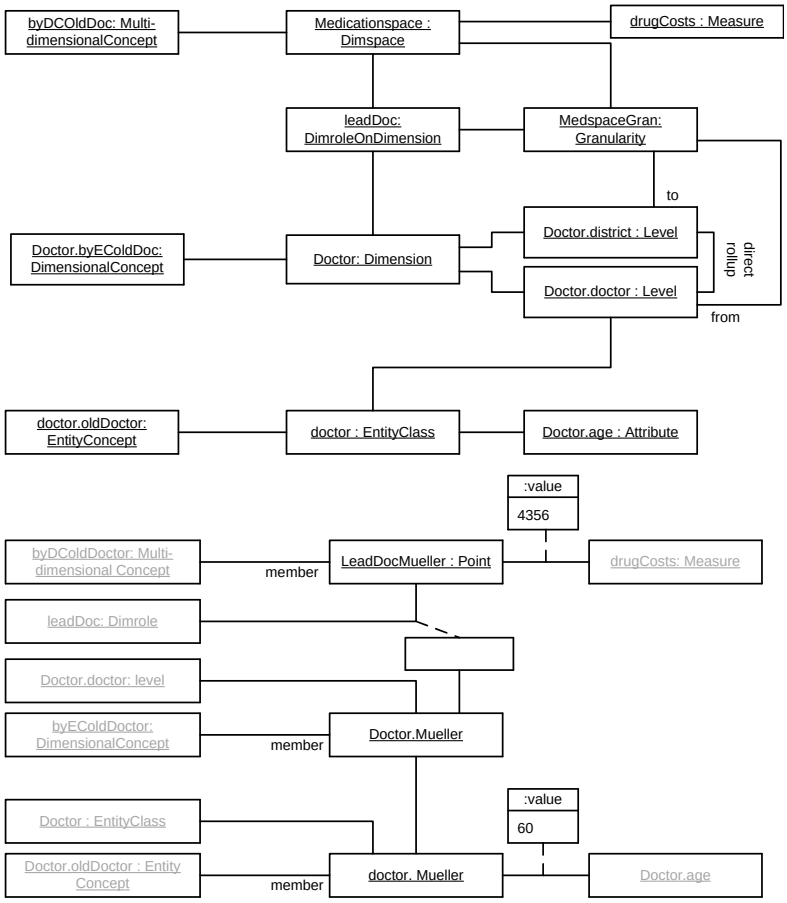

Figure 6: Object diagram for schema perspective (top) and instance perspective (bottom)

```
insert into dw_entityclass( entityclassName) values( 'doctor');
insert into dw_attribute(entityClassId, attributeName, datatype) values( 'doctor', 'age', 'integer');
insert into dw_dimension(dimensionName) values( 'Doctor' );
insert into dw_level( entityClassId, levelName, dimensionID) values( 'doctor', 'doctor', 'Doctor');
insert into dw_level( entityClassId, levelName, dimensionID) values( 'district', 'district',    'Doctor' );
insert into dw_level_directrollup(sub_levelid, sup_levelid) values ('Doctor.doctor', 'Doctor.district');
insert into dw_dimrole(dimroleName, discriminator) values( 'leadDoc', 'dimroleondimension' );
insert into dw_dimroleondimension(dimroleid, dimensionid) values('leadDoc', 'Doctor');
insert into dw_dimspace(dimspaceName) values( 'MedicationSpace' );
insert into dw_dimspace_dimrole(dimspaceID, dimroleID) values( 'MedicationSpace', 'leadDoc' );
insert into dw_granularity(dimspaceid, dimroleid, from_levelid, to_levelid) values ('MedicationSpace', 'leadDoc', '
     Doctor.doctor','Doctor.district');
insert into dw_measureinfactclass(factclassid, etlmeasurename, dimspaceId, numericdatatype) values('drugCosts', '
     costs', 'MedicationSpace', 'float');
```

Listing 5: UML Insert statements for creating the object diagram on the schema level

2.3. Structural specification of MDO concepts

In this section we will introduce the different concept types identified by [Neumayr et al., 2013] and give a description of them. As earlier mentioned we have three main types of concepts entity concepts, dimensional concepts and multi-dimensional concepts, the remainder of this section is structured accordingly to these types.

2.3.1. Entity Concepts

Entity concepts are concepts that are defined over an entity class which acts as the domain for the entity concept. An entity class is the collection of all entities of a type, for example the entity class district contains all districts. Entity concepts consists of a collection of entities that meet the criteria specified in the entity concept definition. Entity concepts can further be divided into two types, primitive and defined entity concepts.

Primitive entity concepts are defined outside the MDO and are not part of the reasoning procedure of this implementation. The two primitive entity concept types are called 'primitive' and 'sql-defined'. The names being self explanatory primitive concepts are defined completely outside the MDO so neither the OWL-based reasoner nor the semantic data warehouse can make any conclusions about this type of concept. Sql-defined concepts are concepts where an SQL query is defined by the user to get certain entities. The SQL query of a sql-defined concept is issued on the semDWH, not the MDO-DB, therefore this concept is a blackbox for the OWL implementation but can be interpreted by the semantic data warehouse.

Defined entity concepts are defined inside the MDO and the reasoner can therefore analyze them. We differentiate between five types of defined entity concepts: nominal concepts, attribute restricted concepts, conjunctive concepts, disjunctve concepts and complement concepts. Nominal concepts are a collection of entities associated with an entity class. Attribute restricted concepts restrict one attribute of an entity class and consist of all entities of this class that fulfill the restriction. Disjunctive entity concepts are a disjunction of one or more entity concepts. Conjunctive entity concepts are a conjunction of one or more entity concepts, called conjunctive terms. Conjunctive terms must be entity concepts of the same entity class. A complement entity concept consists of entities that are the complement of another entity concept.

A depiction of entity concepts is shown in figure 7. Listing 6 shows the DDL statements necessary to create entity concepts in the MDO-DB.

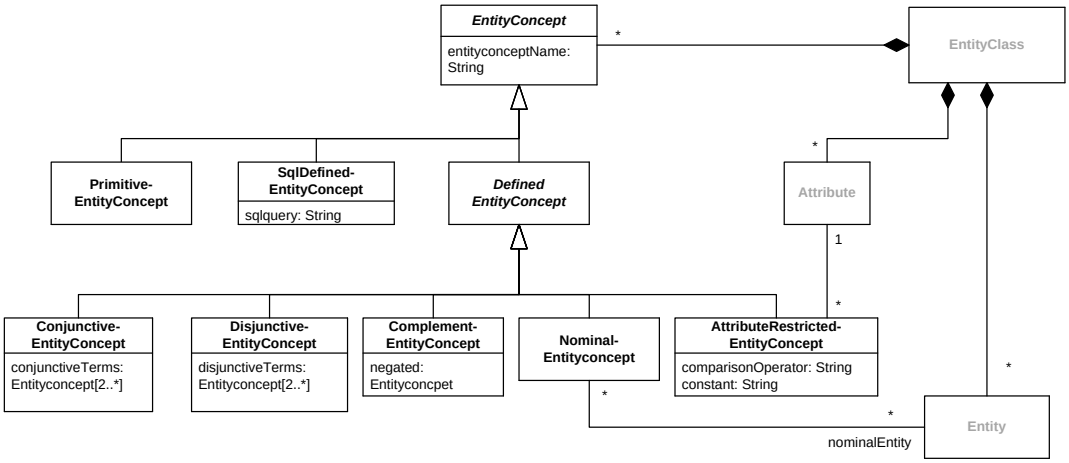

Figure 7: Entity Concept UML representation

```
1  CREATE TABLE ec_entityconcept (
2    ecID varchar2(201)  GENERATED ALWAYS AS (
         entityclassID || '.' || entityconceptName)
         VIRTUAL,
3    entityclassID varchar2(100) not null,
4    entityconceptName varchar2(100) not null,
5    discriminator varchar2(30) not null,
6    CONSTRAINT PRIMARY KEY (ecID),
7    CONSTRAINT FOREIGN KEY (entityclassID)
8    REFERENCES dw_entityclass(entityclassID),
9    CONSTRAINT ec_discriminator CHECK(discriminator
         IN ('primitive','sqldefined',
10   'attributerestriction','byexternalconcept','
         byexternalconceptexpression',
11   'disjunctive','conjunctive','complement','nominal
         '));
12 );
13 CREATE TABLE ec_primitives(
14   ecID varchar2(201) NOT NULL,
15   CONSTRAINT PRIMARY KEY (ecID),
16   CONSTRAINT FOREIGN KEY(ecID)
17   References ec_entityconcept (ecID)
18 );
19 CREATE TABLE ec_sqldefined(
20   ecID varchar2(201) NOT NULL,
21   sqlquery CLOB not null,
22   CONSTRAINT PRIMARY KEY (ecID),
23   CONSTRAINT FOREIGN KEY(ecID)
24   References ec_entityconcept (ecID)
25 );
26 CREATE TABLE ec_attributerestriction(
27   ecID varchar2(201) NOT NULL,
28   attributeID varchar2(100) not null,
29   comparisonoperator varchar2(2) not null,
30   "VALUE" varchar2(100) not null,
31   CONSTRAINT PRIMARY KEY (ecID),
32   CONSTRAINT FOREIGN KEY (attributeID)
33   references dw_attribute(attributeID),
34   CONSTRAINT FOREIGN KEY(ecID)
35   References ec_entityconcept (ecID),
36   CONSTRAINT CHECK(comparisonoperator IN ('<','<=',
         '>','>=','='))
37 );
38 CREATE TABLE ec_disjunctive(
39   ecID varchar2(201) not null,
40   CONSTRAINT PRIMARY KEY (ecID),
41   CONSTRAINT FOREIGN KEY(ecID)
42   References ec_entityconcept (ecID)
43 );
```

```
44 CREATE TABLE ec_disjunctive_term(
45   ecID varchar2(201) not null,
46   term_ecID varchar2(201) not null,
47   CONSTRAINT FOREIGN KEY(term_ecID)
48   REFERENCES ec_entityconcept(ecid),
49   CONSTRAINT FOREIGN KEY(ecID)
50   REFERENCES ec_disjunctive(ecid),
51   CONSTRAINT PRIMARY KEY (ecID,term_ecID)
52 );
53 CREATE TABLE ec_conjunctive(
54   ecID varchar2(201) not null,
55   CONSTRAINT PRIMARY KEY (ecID),
56   CONSTRAINT FOREIGN KEY(ecID)
57   References ec_entityconcept (ecID)
58 );
59 CREATE TABLE ec_conjunctive_term(
60   ecID varchar2(201) not null,
61   term_ecID varchar2(201) not null,
62   CONSTRAINT FOREIGN KEY(term_ecID)
63   REFERENCES ec_entityconcept(ecid),
64   CONSTRAINT FOREIGN KEY(ecID)
65   REFERENCES ec_conjunctive(ecid),
66   CONSTRAINT PRIMARY KEY (ecID,term_ecID)
67 );
68 CREATE TABLE ec_complement(
69   ecID varchar2(201) not null,
70   negated_ecID varchar2(201) not null,
71   CONSTRAINT PRIMARY KEY (ecID),
72   CONSTRAINT FOREIGN KEY(negated_ecID)
73   REFERENCES ec_entityconcept(ecID),
74   CONSTRAINT FOREIGN KEY(ecID)
75   References ec_entityconcept (ecID)
76 );
77 CREATE TABLE ec_nominal(
78  ecID varchar2(201) not null,
79  CONSTRAINT PRIMARY KEY(ecID),
80  CONSTRAINT FOREIGN KEY(ecID)
81   References ec_entityconcept (ecID)
82 );
83 CREATE TABLE ec_nominal_entity(
84  ecID varchar2(201) not null,
85  entityID varchar2(201) not null,
86  CONSTRAINT PRIMARY KEY(ecID,entityID),
87  CONSTRAINT FOREIGN KEY(entityID)
88  references dw_entity(entityID),
89  CONSTRAINT FOREIGN KEY(ecID)
90  references ec_nominal(ecID)
91 );
```

Listing 6: DDL statements for creating entity concepts

The base class of all entity concepts (listing 6 line 1) consists of the name of the entity concept and the entity class for which the entity concept is defined. The resulting name for an entity concept is in the form of 'entityclassID' dot 'conceptName', with this naming pattern the user can at once see for which entity class an entity concept is defined. In our following examples the entityclass id is omitted to keep the names short. The base class also contains a discriminator attribute to distinguish the different entity concept types.

The implementation has a further differentiation between concept types: single table concepts and multi-table concepts. Single table concepts are concepts which consist of only one concept specific table in the MDO-DB. Single table concepts are: Primitive, sql-defined, attribute-restricted and complement entity concepts. As an example see attribute restricted concept definition in listing 6 line 26.

Multi table concepts in contrast to single table concepts consist of more than one concept specific table. Multi table concepts are: Nominal, conjunctive and disjunctive entity concepts. As an example implementation see nominal entity concepts. A nominal entity concept consists of the ec_nominal table (line 77) containing the name of the concept and the ec_nominal_entity table (line 83) containing the entities assigned to the nominal concepts.

This differentiation between single and multi table concept types will be relevant when mapping concepts to OWL (section 4.6.1). Single table concepts and multi table concepts are not specific to entity concepts but also occur at dimensional and multi-dimensional concepts.

Figure 8 depicts the objects diagram for a part of the defined test concepts. The insert statements for all test concepts are shown in listing 7. As already mentioned every entity concept needs at least two inserts, one in the entity concept base table and one in the concept specific table. To keep the listing compact we only show one insert in the entity concept base table, the other inserts in the base table are omitted.

Entity Concept Examples As starting point attribute restricted entity concepts are defined. One attribute restricted entity concept is the concept 'highPopulationDistrict'. This concept has as domain the entity class 'district' and is defined as 'population >80.000' (see listing 7 line 12), all district that have a population greater than 80000 are in the interpretation of this concept. In the same manner the other attribute restricted concepts are defined. The nominal concepts 'myProvince' consists of the entity 'Niederösterreich', keep in mind that an entity concept has exactly one entity class as domain so a mix of provinces and districts would not be allowed. The conjunctive entity concept 'highDensDistrict' consists of the two attribute restricted concepts 'highPopulationDistrict' and 'smallDistrict' (sqkm <150). The conjunctive concept 'YoungOrOldDoctor' consists of the two attribute restricted concepts 'YoungDoctor' (age <35) and 'oldDoctor' (age >60). The concept 'notSmallDistrict' is defined as the complement of 'smallDistrict'. All concept definitions are shown in listing 7.

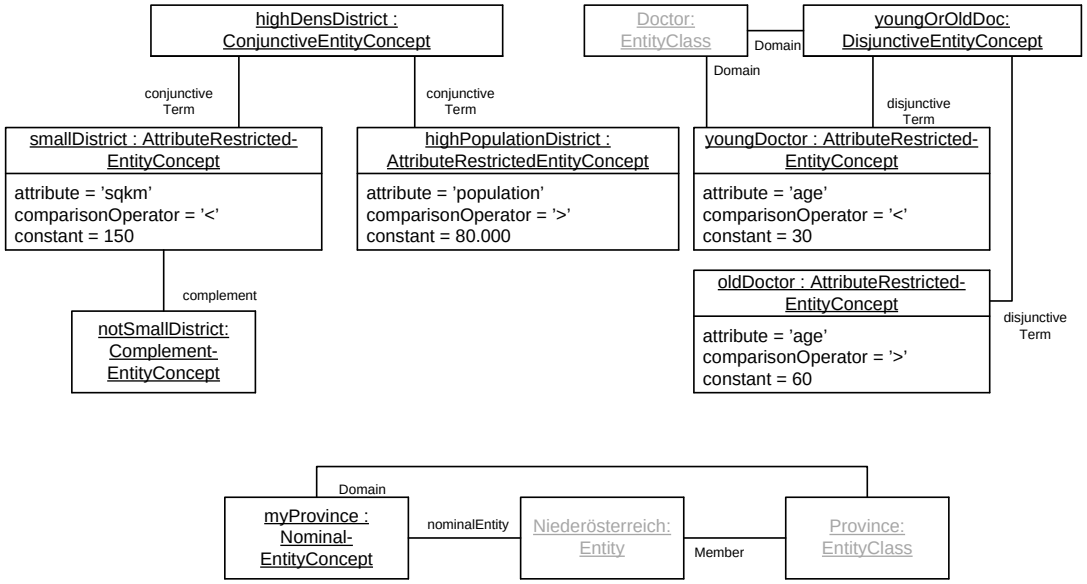

Figure 8: Entity concept object diagram

```
1  INSERT INTO ec_entityconcept (entityclassId,
       entityconceptName, discriminator)
2    VALUES ('doctor', 'YoungDoctor', '
         attributerestriction')
3  INSERT INTO ec_attributerestriction (ecId,
       attributeId, comparisonoperator, "VALUE")
4    VALUES ('doctor.YoungDoctor', 'doctor.age', '<',
         '35')
5
6  INSERT INTO ec_attributerestriction (ecId,
       attributeId, comparisonoperator, "VALUE")
7    VALUES ('doctor.VeryYoungDoctor', 'doctor.age', '
         <', '30')
8
9  INSERT INTO ec_attributerestriction (ecId,
       attributeId, comparisonoperator, "VALUE")
10   VALUES ('doctor.OldDoctor', 'doctor.age', '>', '
         60')
11
12 INSERT INTO ec_attributerestriction (ecId,
       attributeId, comparisonoperator, "VALUE")
13   VALUES ('district.HighPopulationDistrict', '
         district.inhabitants', '>', '80000')
14
15 INSERT INTO ec_attributerestriction (ecId,
       attributeId, comparisonoperator, "VALUE")
16   VALUES ('district.SmallDistrict', 'district.sqkm'
         , '<', '150')
17
18 INSERT INTO ec_nominal (ecId)
19   VALUES ('province.myProvince')
20
21 INSERT INTO ec_nominal_entity (ecId, entityID)
22   VALUES ('province.myProvince', 'province.
         LowerAustria')
23
24 INSERT INTO ec_conjunctive (ecId)
25   VALUES ('district.HighDensDistr')
26
27 INSERT INTO ec_conjunctive_term (ecId, term_ecID)
28   VALUES ('district.HighDensDistr', 'district.
         SmallDistrict')
29 INSERT INTO ec_conjunctive_term (ecId, term_ecID)
30   VALUES ('district.HighDensDistr', 'district.
         HighPopulationDistrict')
31
32 INSERT INTO ec_disjunctive (ecId)
33   VALUES ('doctor.YoungOrOldDoc')
34
35 INSERT INTO ec_disjunctive_term (ecId, term_ecID)
36   VALUES ('doctor.YoungOrOldDoc', 'doctor.
         YoungDoctor')
37 INSERT INTO ec_disjunctive_term (ecId, term_ecID)
38   VALUES ('doctor.YoungOrOldDoc', 'doctor.OldDoctor
         ')
39
40 INSERT INTO ec_complement (ecId, negated_ecID)
41   VALUES ('district.notSmallDistrict', 'district.
         SmallDistrict')
```

Listing 7: DML statements for creating entity concepts

2.3.2. Dimensional Concepts

Dimensional Concepts are defined over a dimension which is the domain for the concept. The resulting dimensional concepts consists of a collection of nodes. A node is at a specific level and references an entity of an entity class. Similar to entity concepts, dimensional concepts are also divided into primitive and defined dimensional concepts. The primitive dimensional concepts are, in the same style as entity concepts, called 'primitive' and 'sql-defined' and share the same behavior.

A new added aspect, compared to entity concepts, is the existence of levels. Every dimension consists of different levels and every node is assigned to a specific level and rolls up to another specific node of a higher level. This aspect adds the 'hierarchy property' to dimensional concepts which states that: 'if a node is in the interpretation of a dimensional concept, then all its sub- and superordinate nodes within the domain of the concept are also in the interpretation of the concept' [Neumayr et al., 2013, p. 11]. The domain over which a dimensional concept is defined is called its signature. The signature of a dimensional concept can either be flat or hierarchical. A flat signature consists of only one level whereas a hierarchical signature spreads from a top level to a bottom level.

There are six different kinds of defined dimensional concepts: dimensional concepts defined by an entity concept, hierarchy expanded concepts, level range restricted concepts, disjunctive concepts, conjunctive concepts and complement concepts. 'By entity concept' dimensional concepts are defined as concepts where the nodes reference the entities defined by the entity concept. Hierarchy expanded concepts take a dimensional concept and expand it so that all direct or indirect successor nodes are in the concept, therefore explicitly implementing the aforementioned hierarchy property but only in the bottom level direction. By level range restriction concepts restrict the nodes of a concept to a certain level or level range, it can therefore be seen as the counterpart to the hierarchy expansion and the hierarchy property. Conjunctive, disjunctive and complement dimensional concepts act similar as their entity counterparts. Disjunctive concepts are a disjunction of one or more dimensional concepts with the restriction that they all must be at the same level range. Conjunctive concepts are a conjunction of one or more dimensional concepts. Complement dimensional concepts are the complement of another dimensional concept where the result of the complement is restricted to the domain of the complement concept. Figure 9 shows the UML diagram representing dimensional concepts. Listing 8 shows the DDL statements for creating dimensional concepts in the MDO-DB. Listing 8 omits the depiction of constraint names, also the definition check constraint of the dimensional concept base class was shortened to show only dimensional concept specific definitions.

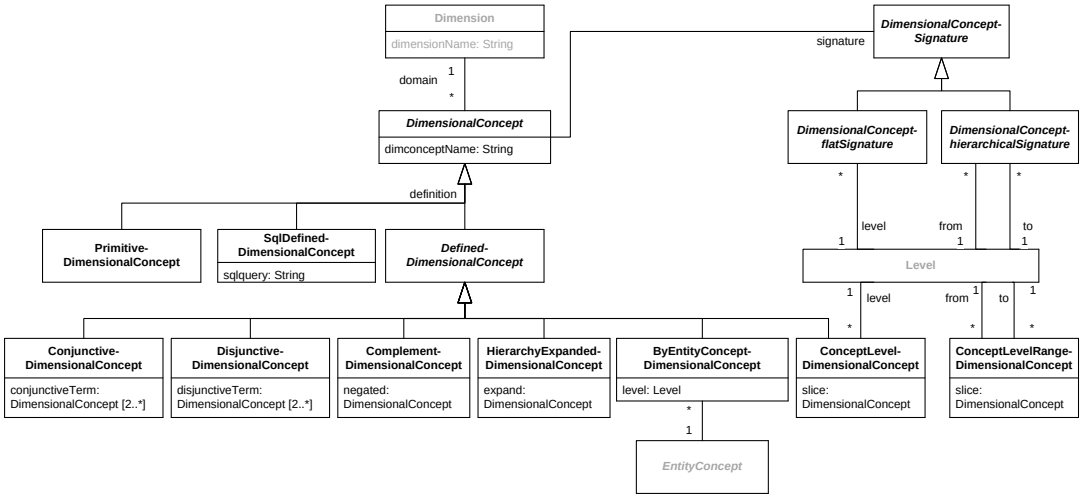

Figure 9: UML class diagram for representing dimensional concepts

```
1  CREATE TABLE dc_dimconcept(
2    dcID varchar2(303) GENERATED ALWAYS AS (
         dimensionID || '.' || dimconceptName)
         VIRTUAL,
3    dimensionID varchar2(100) not null,
4    dimconceptName varchar2(201) not null,
5    signature_discriminator varchar2(30) not null,
6    definition_discriminator varchar2(30) not null
7    CONSTRAINT PRIMARY KEY(dcID),
8    CONSTRAINT  FOREIGN KEY(dimensionID)
9    REFERENCES dw_dimension(dimensionID),
10   CONSTRAINT CHECK(definition_discriminator IN
11   ( 'byentityconcept','hierarchyexpansion',
12  'conceptlevel', 'conceptlevelrange')),
13   CONSTRAINT CHECK(signature_discriminator IN (
14   'flatsignature','hierarchicalsignature')));
15 CREATE TABLE dc_flatSignature(
16     dcID varchar2(201) PRIMARY KEY,
17     levelID varchar2(201) not null,
18     CONSTRAINT FOREIGN KEY(levelID)
19     references dw_level(levelID),
20     CONSTRAINT FOREIGN KEY(dcID)
21     REFERENCES dc_dimconcept(dcID));
22 CREATE TABLE dc_hierarchicalSignature(
23     dcID varchar2(201) PRIMARY KEY,
24     from_levelID varchar2(201) not null,
25     to_levelID varchar2(201) not null,
26     CONSTRAINT FOREIGN KEY(from_levelID)
27     references dw_level(levelID),
28     CONSTRAINT FOREIGN KEY(to_levelID)
29     references dw_level(levelID),
30     CONSTRAINT FOREIGN KEY(dcID)
31     REFERENCES dc_dimconcept(dcID));
32 CREATE TABLE dc_primitive(
33     dcID varchar2(201) PRIMARY KEY);
34 CREATE TABLE dc_sqldefined(
35     dcID varchar2(201) PRIMARY KEY,
36     sqlquery CLOB not null,
37     CONSTRAINT FOREIGN KEY(dcID)
38     references dc_dimconcept(dcID));
39 CREATE TABLE dc_byentityconcept(
40     dcID varchar2(303) PRIMARY KEY,
41     levelID varchar2(201) not null,
42     ecID varchar2(201) not null,
43     CONSTRAINT FOREIGN KEY (levelID)
44     references dw_level(levelID),
45     CONSTRAINT FOREIGN KEY (ecID)
46     references ec_entityconcept(ecID),
47     CONSTRAINT  FOREIGN KEY(dcID)
48     references dc_dimconcept(dcID));
49 CREATE TABLE dc_hierarchyexpansion(
50     dcID varchar2(201) PRIMARY KEY,
51     tobeexpanded_dcID varchar2(303) not null,
52     CONSTRAINT FOREIGN KEY (tobeexpanded_dcID)
53     references dc_dimconcept(dcID));
54 CREATE TABLE dc_conjunctive(
55     dcID varchar2(201) PRIMARY KEY,
56     CONSTRAINT FOREIGN KEY(dcID)
57     references dc_dimconcept(dcID));
58 CREATE TABLE dc_conjunctive_term(
59     dcID varchar2(201) not null,
60     term_dcID varchar2(201) not null,
61     CONSTRAINT PRIMARY KEY(dcID,term_dcID),
62     CONSTRAINT FOREIGN KEY (term_dcID)
63     REFERENCES dc_dimconcept(dcID));
64 CREATE TABLE dc_disjunctive(
65     dcID varchar2(201) PRIMARY KEY,
66     CONSTRAINT FOREIGN KEY(dcID)
67     references dc_dimconcept(dcID));
68 CREATE TABLE dc_disjunctive_term(
69     dcID varchar2(201) not null,
70     term_dcID varchar2(201) not null,
71     CONSTRAINT PRIMARY KEY(dcID,term_dcID),
72     CONSTRAINT FOREIGN KEY (term_dcID)
73     REFERENCES dc_dimconcept(dcID));
74 CREATE TABLE dc_complement(
75     dcID varchar2(201) PRIMARY KEY,
76     negated_dcID varchar2(201) not null,
77     CONSTRAINT FOREIGN KEY (negated_dcID)
78     REFERENCES dc_dimconcept(dcID),
79     CONSTRAINT FOREIGN KEY(dcID)
80     references dc_dimconcept(dcID));
81 CREATE TABLE dc_conceptLevel(
82     dcID varchar2(201) PRIMARY KEY,
83     slice_dcID varchar2(201) not null,
84     levelID varchar2(201) not null,
85     CONSTRAINT FOREIGN KEY(levelID)
86     REFERENCES dw_level(levelID),
87     CONSTRAINT FOREIGN KEY(slice_dcID)
88     REFERENCES dc_dimconcept(dcID));
89 CREATE TABLE dc_conceptLevelRange(
90     dcID varchar2(201) PRIMARY KEY,
91     slice_dcID varchar2(201) not null,
92     from_levelID varchar2(201) not null,
93     to_levelID varchar2(201) not null,
94     CONSTRAINT FOREIGN KEY(from_levelID)
95     REFERENCES dw_level(levelID),
96     CONSTRAINT FOREIGN KEY(to_levelID)
97     REFERENCES dw_level(levelID),
98     CONSTRAINT FOREIGN KEY(slice_dcID)
99     REFERENCES dc_dimconcept(dcID));
```

Listing 8: DDL statements for creating dimensional concepts

Dimensional concepts have a similar structure as entity concepts, again the dimensional concepts consist of a base table (listing 8 line 2) and a concept specific table. In contrast to entity concepts the base table of dimensional concepts also contains a discriminator attribute for the signature(listing 8 line 13) that is defined for the dimensional concept. A flat signature consists of only one level (line15), a hierarchical signature spans from one level to another (line 22).

As with entity concepts the implementation again differentiates between single table concepts (for example by entity concepts dimensional concepts line 39) and multi table concepts (for example conjunctive dimensional concepts line 54).

Dimensional concept examples To be able to work with dimensional concepts, first entity concepts need to be converted into dimensional concepts. For this purpose 'by entity concept' dimensional concepts are used. The two entity concepts 'HighDensDistrict' and 'YoungInsurant' are both transferred into the dimensional concepts 'byECHighDensDis' and 'byECYoungInsurant'. Both concepts are linked to the insurant dimension. 'NotHighDensity' is the complement concept of 'byECHighDensDis' . The entity concept 'HighPopulationDistrict' is transformed into two dimensional concepts, one concept 'DocHPC' for the doctor dimension and one concept 'InsHPC' for the insurant dimension. The dimensional concept 'byECHighDensDis' is expanded so it spans from the insurant level to the district level. The resulting concept 'InHighDensDis' is than restricted to the insurant level which creates the concept level concept 'InsInHighDensDis'. 'InsInHighDensDis' is combined with 'byECYoungInsurant' to create the conjunctive dimensional concept 'YoungInsInHighDens'. The same two terms are used to create the disjunctive concept 'YoungInsOrHighDens'. An sql-defined concept is generated that gets all doctors named 'Mayer', the query is : 'select doctor from d_doctor where doctor = 'Mayer' '. Remember, the query of an sql-defined entity concept is executed in the semantic data warehouse, not in the MDO-Base.

The defined concepts are depicted in figure 10. In the figure signatures are omitted to keep it compact. The inserts for all dimensional concepts are shown in listing 9 (including the missing signatures from figure 10).

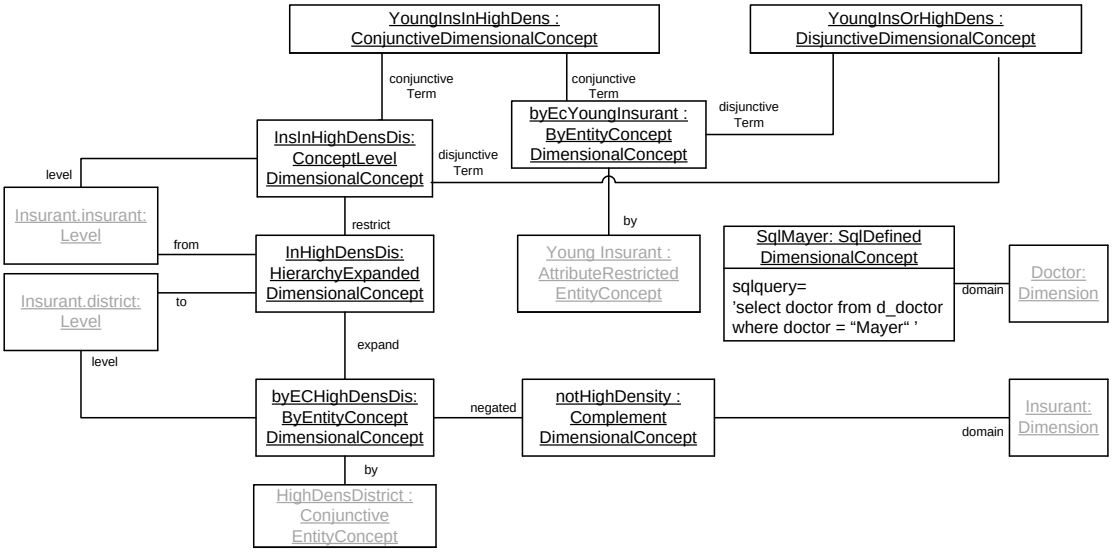

Figure 10: Example dimensional concepts object diagram

```
1  INSERT INTO dc_dimconcept (dimensionId,
       dimconceptName, signature_discriminator,
       definition_discriminator)
2    VALUES ('Doctor', 'SqlMayer', 'flatsignature', '
         sqldefined')
3  INSERT INTO dc_flatSignature (dcID, levelID)
4    VALUES ('Doctor.SqlMayer', 'Doctor.doctor')
5  INSERT INTO dc_sqldefined (dcID, sqlquery)
6    VALUES ('Doctor.SqlMayer', 'select␣doctor␣from␣
         d_doctor␣')
7
8  INSERT INTO dc_flatSignature (dcID, levelID)
9    VALUES ('Insurant.byECHighDensDis', 'Insurant.
         district')
10 INSERT INTO dc_byentityconcept (dcID, levelId, ecId
       )
11   VALUES ('Insurant.byECHighDensDis', 'Insurant.
         district', 'district.HighDensDistr')
12
13 INSERT INTO dc_hierarchicalSignature (dcID,
       from_levelID, to_levelID)
14   VALUES ('Insurant.InHighDensDis', 'Insurant.
         insurant', 'Insurant.district')
15 INSERT INTO dc_hierarchyexpansion (dcID,
       tobeexpanded_dcID)
16   VALUES ('Insurant.InHighDensDis', 'Insurant.
         byECHighDensDis')
17
18 INSERT INTO dc_flatSignature (dcID, levelID)
19   VALUES ('Insurant.InsInHighDensDis', 'Insurant.
         insurant')
20 INSERT INTO dc_conceptLevel (dcID, slice_dcID,
       levelID)
21   VALUES ('Insurant.InsInHighDensDis', 'Insurant.
         InHighDensDis', 'Insurant.insurant')
22
23 INSERT INTO dc_flatSignature (dcID, levelID)
24   VALUES ('Insurant.notHighDensity', 'Insurant.
         district')
25 INSERT INTO dc_complement (dcID, negated_dcID)
26   VALUES ('Insurant.notHighDensity', 'Insurant.
         byECHighDensDis')
27
28 INSERT INTO dc_hierarchicalSignature (dcID,
       from_levelID, to_levelID)
29   VALUES ('Insurant.ExpandnotHighDensity', '
         Insurant.insurant', 'Insurant.district')
30 INSERT INTO dc_hierarchyexpansion (dcID,
       tobeexpanded_dcID)
31   VALUES ('Insurant.ExpandnotHighDensity', '
         Insurant.notHighDensity')
32
33 INSERT INTO dc_flatSignature (dcID, levelID)
34   VALUES ('Insurant.byECYoungInsurant', 'Insurant.
         insurant')
35 INSERT INTO dc_byentityconcept (dcID, levelId, ecId
       )
36   VALUES ('Insurant.byECYoungInsurant', 'Insurant.
         insurant', 'insurant.YoungInsurant')
37
38 INSERT INTO dc_flatSignature (dcID, levelID)
39   VALUES ('Insurant.YoungInsInHighDens', 'Insurant.
         insurant')
40 INSERT INTO dc_conjunctive (dcID)
41   VALUES ('Insurant.YoungInsInHighDens')
42 INSERT INTO dc_conjunctive_term (dcID, term_dcID)
43   VALUES ('Insurant.YoungInsInHighDens', 'Insurant.
         InsInHighDensDis')
44 INSERT INTO dc_conjunctive_term (dcID, term_dcID)
45   VALUES ('Insurant.YoungInsInHighDens', 'Insurant.
         byECYoungInsurant')
46
47 INSERT INTO dc_flatSignature (dcID, levelID)
48   VALUES ('Insurant.YoungInsOrHighDens', 'Insurant.
         insurant')
49 INSERT INTO dc_disjunctive (dcID)
50   VALUES ('Insurant.YoungInsOrHighDens')
51 INSERT INTO dc_disjunctive_term (dcID, term_dcID)
52   VALUES ('Insurant.YoungInsOrHighDens', 'Insurant.
         InsInHighDensDis')
53 INSERT INTO dc_disjunctive_term (dcID, term_dcID)
54   VALUES ('Insurant.YoungInsOrHighDens', 'Insurant.
         byECYoungInsurant')
```

Listing 9: DML statements for creating dimensional concepts

2.3.3. Multi-Dimensional Concepts

Multi-dimensional concepts are defined over a dimension space which is the domain for the concept. A dimension space consists of multiple dimensions. A dimension can be represented in a dimension space multiple times because of dimension roles, a dimension role is the role of the dimension it plays in the concrete dimension space. For example we could have the dimension 'doctor' and for this dimension we could define the dimension roles 'acting doctor' and 'prescribing doctor', in this case the same dimension is represented twice in one dimension space. The multi-dimensional concepts consist of a collection of points. A point is defined for a dimension space and consists of multiple nodes where one node references exactly one dimension role. Again, similar to entity concepts and dimensional concepts, multi-dimensional concepts are divided into pimitive and defined concepts. A new aspect of multi-dimensional concepts is that defined concepts are again divided into dimension based and fact based multi dimensional concepts.

Analogous to the level extension of dimensional concepts, multi-dimensional concepts have a granularity extension. The granularity defines over which different levels at the different dimension roles the dimensionspace spans. An example granularity would be acting Doctor from doctor to district and prescribing doctor from doctor to province.

Primitive multi-dimensional concepts are again called 'primitve' and 'sql-defined' and behave like their respective entity or dimensional counterparts.

Dimension based multi-dimensional concepts are referencing only dimensional concepts. There are six kinds of dimension based multi-dimensional concepts: multi-dimensional concepts defined by a dimensional concept, hierarchy expanded concepts, granularity restricted concepts, conjunctive concepts, disjunctive concepts and complement concepts. 'By dimensional concept' multi-dimensional concepts is a concept that references a dimensional concept for a dimensional role. All points referencing this dimensional role are in the multi-dimensional concept. Hierarchy expanded concepts reference a multi-dimensional concept and all points that are descendants of the referenced point are in this concept. Granularity restricted concepts restrict another multi-dimensional concept to a smaller granularity range. Disjunctive concepts are a disjunction of one or more multi-dimensional concepts of the same domain. Conjunctive concepts are a conjunction of one ore more multi-dimensional concepts. Complement concepts consist of the complementing points of another multi-dimensional concepts.

Fact based multi-dimensional concepts are defined over predicates on measure values. We have only one kind of fact based multi-dimensional concept and this kind of concept is also called fact based multi dimensional concept. Fact based multi dimensional concepts can be seen similar to attribute restricted entity concepts only that they do not restrict the attribute of an entity class but the fact values of a measure. Fact based multi-dimensional concepts where not implemented and are therefore not covered in this thesis. For a more thorough description of fact based concepts see [Neuböck et al., 2014, p.15].

Figure 11 shows the UML diagram for multidimensional concepts. Listing 10 shows the DDL statements for creating multi-dimensional concepts in the MDO-DB. Listing 10 lines 12 and 20 show the signature for multi-dimensional concepts. Like dimensional

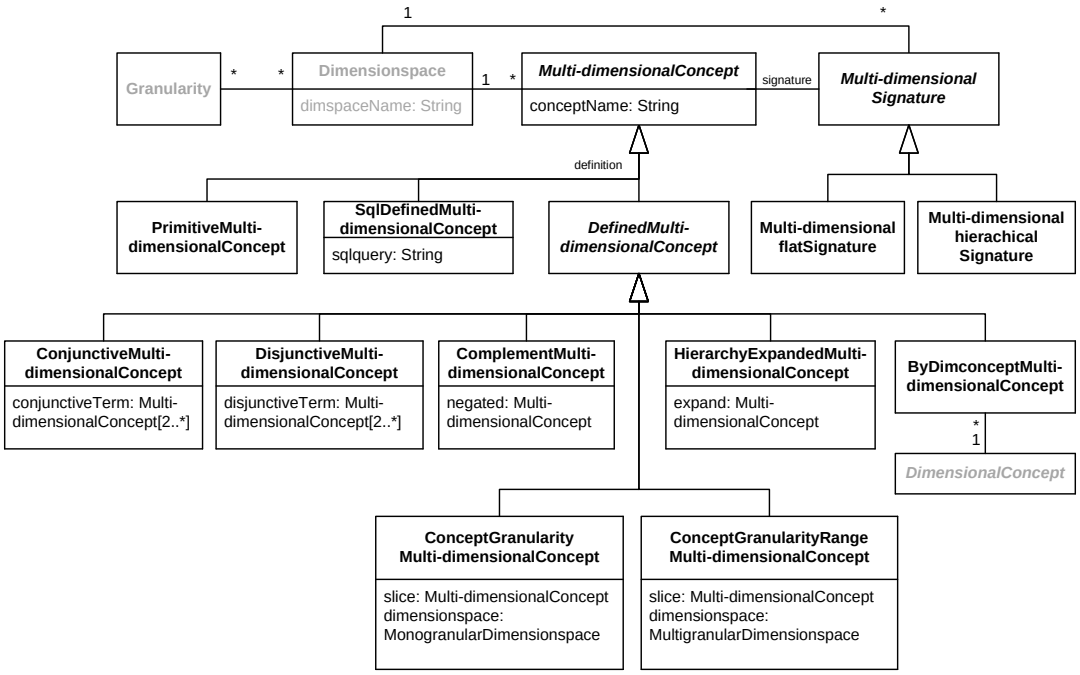

Figure 11: UML class diagram for representing multi-dimensional concepts

concept, multi-dimensional concepts may have a flat signature or a hierarchical signature. A flat signature of a multi-dimensional concept refers to a monogranular dimension space. The individual dimension roles of a monogranular dimension space are restricted to exactly one dimension level, in contrast to that is the multigranular dimension space whose dimension roles can be restricted to an arbitrary number of dimension levels. Hierarchcal signatures refer to multigranular dimension spaces.

Multi-dimensional concepts have again a similar structure as entity concepts and dimensional concepts. The schema for multi-dimensional concepts consist of a base table (listing 10 line 2) and a concept specific table for each kind of concept. Further multi-dimensional concepts can be single- or multi table concepts.

'Concept granularity' concepts restrict a target concept to a monogranular dimension space (listing 10 line 93) whereas 'concept granularityrange' concepts restrict a concept to a multigranular dimension space (listing 10 line 105).

```
CREATE TABLE mdc_mdconcept(
  mdcID varchar2(201) GENERATED ALWAYS AS (
       mdconceptName || '') VIRTUAL,
  mdconceptName varchar2(201) not null,
  signature_discriminator varchar2(30) not null,
  definition_discriminator varchar2(30) not null
  CONSTRAINT mdc_mdconcept_pk PRIMARY KEY (mdcID),
  CONSTRAINT CHECK(definition_discriminator IN (
     'bydimconcept', 'conceptgranularity',  '
          conceptgranularityrange' )),
  CONSTRAINT CHECK(signature_discriminator IN (
  'flatsignature','hierarchicalsignature'))
);
CREATE TABLE mdc_flatSignature(
  mdcID varchar2(100) PRIMARY KEY,
  dimspaceID varchar2(100) not null,
  CONSTRAINT FOREIGN KEY (dimspaceID)
  REFERENCES dw_monogranulardimspace(dimspaceID),
  CONSTRAINTFOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
CREATE TABLE mdc_hierarchicalSignature(
  mdcID varchar2(100) PRIMARY KEY,
  dimspaceID varchar2(100) not null,
  CONSTRAINT FOREIGN KEY (dimspaceID)
  REFERENCES dw_multigranularDimspace(dimspaceID),
  CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
CREATE TABLE mdc_primitive(
  mdcID varchar2(100) PRIMARY KEY,
  CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID));
CREATE TABLE mdc_sqldefined(
  mdcID varchar2(100) PRIMARY KEY,
  sqlquery CLOB not null,
  CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
CREATE TABLE mdc_bydimconcept(
  mdcID varchar2(201) PRIMARY KEY,
  dimroleID varchar2(100) not null,
  dcID varchar2(201) not null,
  CONSTRAINTFOREIGN KEY (dimroleID)
  REFERENCES dw_dimrole(dimroleID),
  CONSTRAINT FOREIGN KEY (dcID)
  REFERENCES dc_dimconcept(dcID),
  CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
CREATE TABLE mdc_hierarchyexpansion(
  mdcID varchar2(100) PRIMARY KEY ,
  tobeexpanded_mdcID varchar2(100) not null,
  CONSTRAINT FOREIGN KEY (tobeexpanded_mdcID)
  REFERENCES mdc_mdconcept (mdcID),
   CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
CREATE TABLE mdc_conjunctive(
  mdcID varchar2(100) PRIMARY KEY,
   CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
CREATE TABLE mdc_conjunctive_term(
  mdcID varchar2(100) PRIMARY KEY,
  term_mdcID varchar2(100) not null,
  CONSTRAINT FOREIGN KEY (term_mdcID)
  REFERENCES mdc_mdconcept(mdcID),
  CONSTRAINT FOREIGN KEY (mdcID)
  REFERENCES mdc_conjunctive(mdcID)
);
CREATE TABLE mdc_disjunctive(
  mdcID varchar2(100) PRIMARY KEY,
  CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
CREATE TABLE mdc_disjunctive_term(
  mdcID varchar2(100),
  term_mdcID varchar2(100) not null,
  CONSTRAINT  PRIMARY KEY (mdcID, term_mdcID),
  CONSTRAINT  FOREIGN KEY (term_mdcID)
  REFERENCES mdc_mdconcept(mdcID),
  CONSTRAINT  FOREIGN KEY (mdcID)
  REFERENCES mdc_disjunctive(mdcID)
);
CREATE TABLE mdc_complement(
  mdcID varchar2(100),
  negated_mdcID varchar2(100) not null,
  CONSTRAINT PRIMARY KEY (mdcID),
  CONSTRAINT FOREIGN KEY (negated_mdcID)
  REFERENCES mdc_mdconcept(mdcID),
  CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
CREATE TABLE mdc_conceptGranularityMDC(
  mdcID varchar2(100) ,
  slice_mdcID varchar2(100) not null,
  dimspaceID varchar2(100) not null,
  CONSTRAINT PRIMARY KEY (mdcID),
  CONSTRAINT FOREIGN KEY (slice_mdcID)
  REFERENCES mdc_mdconcept(mdcID),
  CONSTRAINT FOREIGN KEY (dimspaceID)
  REFERENCES dw_monogranulardimspace(dimspaceID),
  CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
CREATE TABLE mdc_conceptGranRangeMDC(
  mdcID varchar2(100),
  slice_mdcID varchar2(100) not null,
  dimspaceID varchar2(100) not null,
  CONSTRAINT PRIMARY KEY (mdcID),
  CONSTRAINT FOREIGN KEY (slice_mdcID)
  REFERENCES mdc_mdconcept(mdcID),
  CONSTRAINT FOREIGN KEY (dimspaceID)
  REFERENCES dw_multigranularDimspace(dimspaceID),
  CONSTRAINT FOREIGN KEY(mdcID)
  REFERENCES mdc_mdconcept(mdcID)
);
```

Listing 10: DDL statements for creating multi-dimensional concepts

Multi-dimensional concept examples Similar to dimensional concepts, first some dimensional concepts have to be transformed into multi-dimensional concepts. For this transformation the 'by dimensional concept' multi-dimensional concept is used (listing 11 line 38). Four concepts are created this way. Out of the four, two concepts 'byDcDocHPC' and 'byDcInsHPC' are joined together to form the conjunctive concept 'DocInsHPC' this concept is the cartesian product of all high population districts of the doctor dimension and high population districts of the insurant dimension. The conjunctive concept is then extended to the individuals level so the resulting concept 'expandDocInsHPC' additionally contains all doctors and insurants that reside inside a high population district. After restricting the expanded concept to a monogranular dimension space on the individuals level, the resulting concept 'restDocInsHPC' contains only the individuals without the districts. With the remaining two 'by dimensional concepts' the disjunctive concept 'YoungOrHighDensInsurants' is created which contains all insurants that are young or reside in a high density district. Also a concept 'notHighDensity' is defined which contains all insurants that do not reside inside a high density restrict. Figure 12 shows the object diagram for the example multi-dimensional concepts, to keep the figure clear some concepts and granularities are omitted. Listing 11 contains all DML statements to create the concepts. Again the base class is only depicted once to keep the listing short.

2.4. Discussion

In this section we covered the structure of the MDO-DB and showed how we created the running example. The MDO-DB need not be implemented using a relational database system, we could also have implemented the multi-dimensional ontology using a object oriented programming language. In fact this approach was used in a first version of the Semantic Cockpit approach, see [Neumayr et al., 2012]. The reason we chose to implement the new version with a database system was that with the database system we could use the built-in integrity checks and multi-user control capabilities.

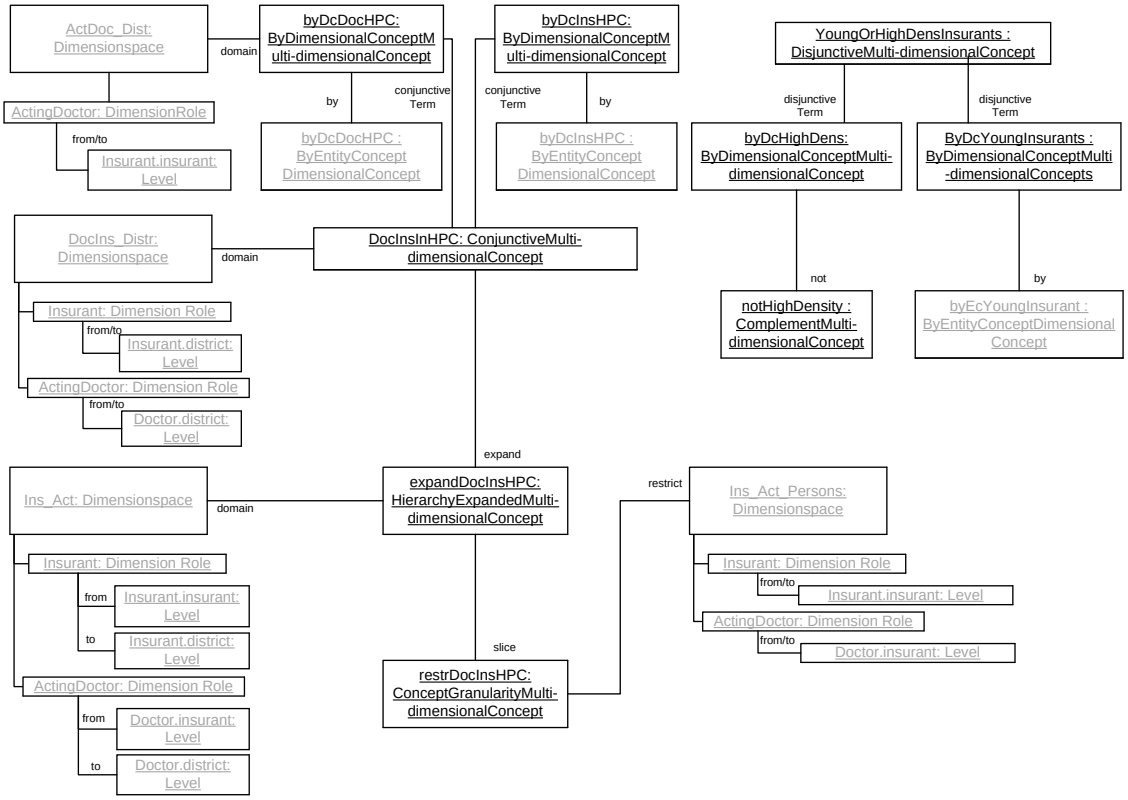

Figure 12: Multi-dimensional concepts object diagram

```
1  INSERT INTO mdc_mdconcept (mdconceptName,
       signature_discriminator,
       definition_discriminator)
2    VALUES ('byDcHighDens', 'flatsignature', '
         bydimconcept')
3  INSERT INTO mdc_flatSignature (mdcId, dimspaceId)
4    VALUES ('byDcHighDens', 'Ins_Distr')
5  INSERT INTO mdc_bydimconcept (mdcid, dimroleid,
       dcid)
6    VALUES ('byDcHighDens', 'insurant', 'Insurant.
         byECHighDensDis')
7  INSERT INTO mdc_flatSignature (mdcId, dimspaceId)
8    VALUES ('byDcYoungIns', 'Ins_Insurants')
9  INSERT INTO mdc_bydimconcept (mdcid, dimroleid,
       dcid)
10   VALUES ('byDcYoungIns', 'insurant', 'Insurant.
         byECYoungInsurant')
11 INSERT INTO mdc_flatSignature (mdcId, dimspaceId)
12   VALUES ('byDcDocHPC', 'ActDoc_District')
13 INSERT INTO mdc_bydimconcept (mdcid, dimroleid,
       dcid)
14   VALUES ('byDcDocHPC', 'actDoc', 'Doctor.DocHPC')
15 INSERT INTO mdc_flatSignature (mdcId, dimspaceId)
16   VALUES ('byDcInsHPC', 'Ins_Distr')
17 INSERT INTO mdc_bydimconcept (mdcid, dimroleid,
       dcid)
18   VALUES ('byDcInsHPC', 'insurant', 'Insurant.
         InsHPC')
19 INSERT INTO mdc_flatSignature (mdcId, dimspaceId)
20   VALUES ('DocInsInHPC2', 'DocIns_Distr')
21 INSERT INTO mdc_conjunctive (mdcID)
22   VALUES ('DocInsInHPC2')
23 INSERT INTO mdc_conjunctive_term (mdcID, term_mdcID
       )
24   VALUES ('DocInsInHPC2', 'byDcInsHPC')
25 INSERT INTO mdc_conjunctive_term (mdcID, term_mdcID
       )
26   VALUES ('DocInsInHPC2', 'byDcDocHPC')
27 INSERT INTO mdc_hierarchicalSignature (mdcId,
       dimspaceId)
28   VALUES ('expandDocInsHPC', 'Ins_Act')
29 INSERT INTO mdc_hierarchyexpansion (mdcID,
       tobeexpanded_mdcID)
30   VALUES ('expandDocInsHPC', 'DocInsInHPC2')
31 INSERT INTO mdc_flatSignature (mdcId, dimspaceId)
32   VALUES ('restrDocInsHPC', 'Ins_Act_persons')
33 INSERT INTO mdc_conceptGranularityMDC (mdcID,
       slice_mdcID, dimspaceID)
34   VALUES ('restrDocInsHPC', 'expandDocInsHPC', '
         Ins_Act_persons')
35 INSERT INTO mdc_flatSignature (mdcId, dimspaceId)
36   VALUES ('YoungOrHighDensInsurants', '
         Ins_Insurants')
37 INSERT INTO mdc_disjunctive (mdcID)
38   VALUES ('YoungOrHighDensInsurants')
39 INSERT INTO mdc_disjunctive_term (mdcID, term_mdcID
       )
40   VALUES ('YoungOrHighDensInsurants', '
         restHighDensInsurants')
41 INSERT INTO mdc_disjunctive_term (mdcID, term_mdcID
       )
42   VALUES ('YoungOrHighDensInsurants', 'byDcYoungIns
         ')
```

Listing 11: DML statements for creating multi-dimensional concepts

3. MDO Parser: ANTLR-based implementation of a concrete MDO syntax

ANTLR is a parser generator we used for defining a grammar for the MDO language definition. With the MDO language the user can create various types of concepts without needing to know the underlying data structure of the concepts(described in section 2.3).

To create the command line interface we used the Java implementation of ANTLR. With the grammar for our concrete MDO syntax defined, ANTLR creates a parser, using the listener pattern, and allows us to use the language in different applications.

This section is structured as follows. First in section 3.1 we give a brief explanation what ANTLR is and how it operates. In the sections 3.2 to 3.5 we will show how the MDO syntax of OLAP cube individuals and the MDO syntax of the concepts is mapped to DML statements via the MDO Parser. Last, in section 3.6 we will see implementation details for the MDO-Parser and discuss the parsers performance.

The initial version of the grammar and the ANTLR implementation was developed by Arjol Qeleshi, a member of the SemCockpit Project.

3.1. Overview

ANTLR stands for Another Tool for Language Recognition and is a parser generator written by Terrence Parr[4]. Basically ANTLR works in the way following way: It takes a grammar, the ANTLR grammar, and from this grammar it creates a parser and a lexer. The ANTLR grammar is very similar to extended Backus-Naur form with some ANTLR specific operations. Lexer and Parser work together to recognize the language input we put into the user interface. A Lexer is a tool that reads an input stream and breaks the input stream into separate junks, so called tokens, this process is called lexical analysis or tokenizing. The tokens are then grouped into token types. In our implementation we have token classes for identifiers or for digits. This token stream is then analyzed by the parser. The parser receives the token stream and creates a parse tree. With this parse tree the parser tries to match the token stream to specific parser rules. For an example mapping process see figure 13.

In this example the user wants to create an attribute restricted entity concept 'Young-Doctor' that comprises all doctors that are younger than 35 years. The lexer takes the input of the user and creates a token stream. This token stream is than matched by the parser to the parser rule for creating an attribute restricted concept and creates a parse tree. The Java implementation then walks through the parse tree and creates a DML statement which is used to create the attribute restricted entity concept in the MDO-DB.

Grammar As mentioned before an ANTLR grammar is very similar to an EBNF grammar. The grammar differentiates between lexer rules and parser rules. As their respective

[4]For a thorough description of ANTLR see [Parr, 2013]

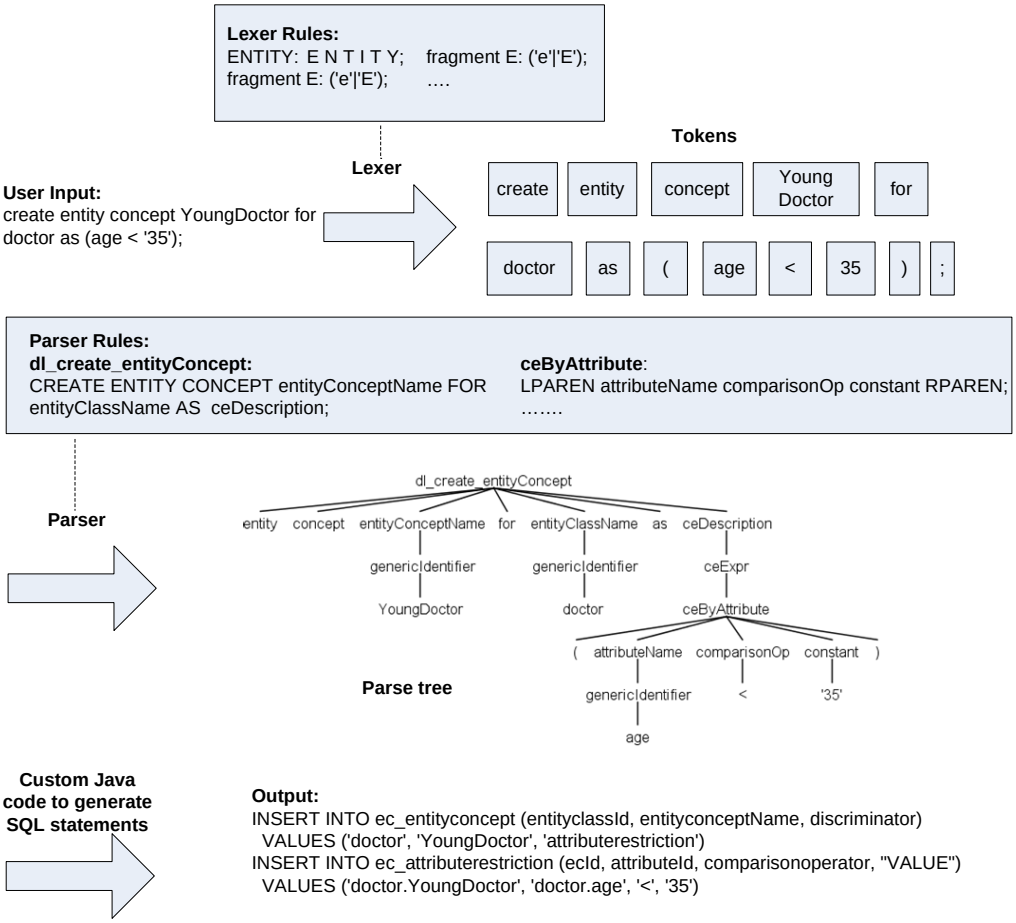

Figure 13: Language recognition example

names state, lexer rules are rules for the lexer to assign the tokens he finds into token types. In ANTLR grammar lexer rules are written in upper case letters. Parser rules are rules for the parser to find the statements we seek in a language pattern. In ANTLR parser rules are written in lower case letters. in figure 13 we have examples for parser rules and for lexer rules.

Next to the parse tree in figure 13 we can see the parser rule 'dl_create_entityConcept'. This rule is responsible for recognizing if the user wants to create a new entity concept. Every concept type has its own parser rule inside the grammar, for our example attribute restricted entity concept we need the parser rule 'ceByAttribute'. For the parser to be able to recognize the different token types we need to define different lexer rules. To recognize the token 'ENTITY' we created the lexer rule 'ENTITY'. 'ENTITY' itself consists of different lexer rules for every letter to make the tokens case insensitive. The keyword 'fragment' we see in the grammar is a particularity of the ANTLR grammar. This keyword tells the lexer that the rule 'E' for the letter is not a token on its own, but only a fragment of another token.

The resulting parse tree of this concept can be seen in figure 13. In the parse tree every subtree can be matched to a rule and the leaves are either symbols or tokens. The whole grammar of the implementation can be found in Appendix B.

3.2. Individuals in OLAP cubes

We did not implement all capabilities for creating an OLAP cube representation into ANTLR because we assumed that the user will most likely interact with an MDO-DB and semDWH where the OLAP parts already exist. For example, entering all entities of a data warehouse through our user interface would theoretically be possible but not suitable. We would advise to write a bulk loading process doing this task, which would be much more suitable. We implemented the creation of nodes and entities because they are necessary for creating different concepts. If we want to extend the capabilities in the future we can easily do so, as we only need to extend our defined grammar.

Table 1 shows the abstract MDO syntax representation and the aspired DML statements for these OLAP cube parts. Table 2 shows example mappings of these OLAP constructs. The node name consists of the dimension name and the entity name, separated by a dot. To create the node we need to use the keyword 'entitynode' because in the course of implementation we planned to support another node type, concept nodes. Concept nodes in contrast to entity nodes do not reference an entity but a concept. As concept nodes where not implemented we do not discuss them further, for a broader explanation of concept nodes see [Neuböck et al., 2014, p.18].

MDO Syntax	**DML statements**
CREATE ENTITY *e* FOR *ecl*;	INSERT INTO dw_entity (entityclassid, entityname) VALUES (*ecl*, *e*)
CREATE ENTITYNODE FOR *d* ENTITY *e* AT *l*	INSERT INTO dw_entitynode (nodeID, dimensionID, entityID, levelID) VALUES (*d* '.' *e*, *d* , *e* , *l*)

Table 1: MDO Syntax for Individuals

MDO Syntax	**Generated SQL (MDO DB)**
create entity Niederösterreich for province;	INSERT INTO dw_entity (entityclassid, entityname) VALUES ('province', 'Niederösterreich')
create entitynode for dimension Doctor entity province.Niederösterreich at Doctor.province	INSERT INTO dw_entitynode (nodeID, dimensionID, entityID, levelID) VALUES ('Doctor.province.Niederösterreich','Doctor', 'province.Niederösterreich', 'Doctor.province')

Table 2: Example Individuals in MDO Syntax

In the next section we will look at how the ANTLR user interface handles the inputs from the user for the different concept types.

3.3. Entity Concepts

In this section we will see how the ANTLR interface handles the different concept types and what insert statements are being created. Table 3 shows the MDO Syntax and the aspired DML statements.

MDO Syntax	**DML statements**
CREATE ENTITY CONCEPT *ec* FOR *ecl* AS	INSERT INTO ec_entityconcept (entityclassId, entityconceptName, discriminator) VALUES (*ecl*,*ec*,*type*)
PRIMITIVE	INSERT INTO ec_primitive (ecId) VALUES (*ec*)
SQL *query*	INSERT INTO ec_sqldefined(ecId, sqlquery) VALUES (*ec*, *query*)
{$entity_1$, ..., $entity_n$}	INSERT INTO ec_nominal (ecId) VALUES (*ec*) INSERT INTO ec_nominal_entity (ecId, entityID) VALUES (*ec*,$entity_1$) ... INSERT INTO ec_nominal_entity (ecId, entityID) VALUES (*ec*,$entity_n$)
attr θ *value*	INSERT INTO ec_attributerestriction (ecId, attributeId, comparisonoperator, "VALUE") VALUES (*ec*, *attr*, θ, *value*)
(ec_1 AND... AND ec_n)	INSERT INTO ec_conjunctive (ecId) VALUES (*ec*) INSERT INTO ec_conjunctive_term (ecId, term_ecID) VALUES (*ec*,ec_1) ... INSERT INTO ec_conjunctive_term (ecId, term_ecID) VALUES (*ec*,ec_n)
(ec_1 OR... OR ec_n)	INSERT INTO ec_disjunctive (ecId) VALUES (*ec*) INSERT INTO ec_disjunctive_term (ecId, term_ecID) VALUES (*ec*,ec_1) ... INSERT INTO ec_disjunctive_term (ecId, term_ecID) VALUES (*ec*,ec_n)
(NOT ec')	INSERT INTO ec_complement (ecId, negated_ecID) VALUES (*ec*,ec')

Table 3: Entity Concept MDO Syntax and resulting SQL inserts into the MDO DB

The type of an entity concept does not need to be stated explicitly, if the user enters a concept definition in a valid form, the parser can derive which concept type he has to use.

In table 3 we can see the impact of multi table and single table concepts. For conjunctive concepts, disjunctive concepts and nominal concepts the parser implementation needs to create insert statements for two tables e.g. one time for the ec_conjunctive table to create the conjunctive concept and one insert statement per conjunctive term for the ec_conjunctive term table.

Table 4 shows the SQL statements created for our running example. To keep the table compact only one concept of every concept type is shown and only the insert into the concept specific table[5]. What we can notice in the insert statements in table 4 are the key values for our concepts. As already mentioned the name of a concept consists of the entity class name and the concept name separated by a dot. With this naming pattern the user can immediately see the domain of an entity concept.

[5]For all commands issued see Appendix A

MDO Syntax	**Generated SQL (MDO DB)**
create entity concept YoungDoctor for doctor as (doctor.age <'35')	INSERT INTO ec_attributerestriction (ecId, attributeId, comparisonoperator, "VALUE") VALUES ('doctor.YoungDoctor', 'doctor.age', '<', '35')
create entity concept myProvince for province as {province.Niederösterreich}	INSERT INTO ec_nominal (ecId) VALUES ('province.myProvince'); INSERT INTO ec_nominal_entity (ecId, entityID) VALUES ('province.myProvince', 'province.Niederösterreich');
create entity concept HighDensDistr for district as (district.High-PopulationDistrict and district.SmallDistrict)	INSERT INTO ec_conjunctive (ecId) VALUES ('district.HighDensDistr') INSERT INTO ec_conjunctive_term (ecId, term_ecID) VALUES ('district.HighDensDistr', 'district.SmallDistrict') INSERT INTO ec_conjunctive_term (ecId, term_ecID) VALUES ('district.HighDensDistr', 'district.HighPopulationDistrict')
create entity concept YoungOrOldDoc for doctor as (doctor.OldDoctor or doctor.YoungDoctor)	INSERT INTO ec_disjunctive (ecId) VALUES ('doctor.YoungOrOldDoc') INSERT INTO ec_disjunctive_term (ecId, term_ecID) VALUES ('doctor.YoungOrOldDoc', 'doctor.YoungDoctor') INSERT INTO ec_disjunctive_term (ecId, term_ecID) VALUES ('doctor.YoungOrOldDoc', 'doctor.OldDoctor')
create entity concept notSmallDistrict for district as (NOT district.SmallDistrict)	INSERT INTO ec_complement (ecId, negated_ecID) VALUES ('district.notSmallDistrict', 'district.SmallDistrict')

Table 4: MDO Syntax and insert examples for entity concepts

3.4. Dimensional Concepts

Dimensional concepts are very similar to entity concepts, Table 5 shows the MDO Syntax and the corresponding DML statements. An added factor being the new level range that needs to be stated when defining a concept.

MDO Syntax	**DML statements**
CREATE DIMENSIONAL CONCEPT *dc* FOR *d* AT $fromLevel$[.. *toLevel*] AS	INSERT INTO dc_dimconcept (dimensionId, dimconceptName, signature_discriminator, definition_discriminator) VALUES (*dim*, *dc*, *signature*, *type*) INSERT INTO dc_flat/hierachicalSignature (dcID, from_level [, to_level]) VALUES (*dc*, $fromlevel$[,*toLevel*])
PRIMITIVE	INSERT INTO dc_primitive (dcId) VALUES (*dc*)
SQL *query*	INSERT INTO dc_sqldefined(dcId, sqlquery) VALUES (*dc*, *query*)
->*level* : *ec*	INSERT INTO dc_byentityconcept (dcID, levelId, ecId) VALUES (*dc*, *level*,*ec*)
EXPAND dc'	INSERT INTO dc_hierarchyexpansion (dcID, tobeexpanded_dcID) VALUES (*dc*,dc')
dc' *level*	INSERT INTO dc_conceptLevel (dcID, slice_dcID, levelID) VALUES (*dc*, dc', *level*)
(dc_1 AND ... AND dc_n)	INSERT INTO dc_conjunctive (dcId) VALUES (*dc*) INSERT INTO dc_conjunctive_term (dcId, term_dcID) VALUES (*dc*,dc_1) ... INSERT INTO dc_conjunctive_term (dcId, term_dcID) VALUES (*dc*,dc_n)
(dc_1 OR ... OR dc_n)	INSERT INTO dc_disjunctive (dcId) VALUES (*dc*) INSERT INTO dc_disjunctive_term (dcId, term_dcID) VALUES (*dc*,dc_1) ... INSERT INTO dc_disjunctive_term (dcId, term_dcID) VALUES (*dc*,dc_n)
(NOT dc')	INSERT INTO dc_complement (dcId, negated_dcID) VALUES (*dc*,dc')

Table 5: Dimensional Concept MDO Syntax and resulting SQL inserts into the MDO DB

The type of the dimensional concept does not need to be stated explicitly, if the user enters a concept definition in a valid form, the parser can derive which concept type he has to use. As described in section 2.3.2 dimensional concepts have a signature and this signature can be flat or hierarchical. The user has to define the levels for which a concept is valid. If the user wants to define a flat concept the user has to state only one level name, if the user wants to define a level range he or she must indicate this by specifying two levels, separated by two dots. Depending on the number of levels stated at the concept definition the parser creates DML statements for either the flatSignature table or the hierarchicalSignature table.

Table 6 shows the Input for our running example and the created output. Table 6 line 1 shows an example definition for a flat dimensional concept and line 3 an example

of a hierarchical concept. To keep the table compact we only show one concept of every concept type and only the insert into the concept specific table[6].

MDO Syntax	**Generated SQL (MDO DB)**
create dimensional concept SqlMayer for Doctor at Doctor.doctor AS sql "select doctor from d_doctor where doctor = 'Mayer'"	INSERT INTO dc_sqldefined (dcID, sqlquery) VALUES ('Doctor.SqlMayer', 'select doctor from d_doctor where doctor = 'Mayer' ')
create dimensional concept byECHighDensDis for Insurant AT Insurant.district AS - >(Insurant.district:district.HighDensDistr)	INSERT INTO dc_byentityconcept (dcID, levelId, ecId) VALUES ('Insurant.byECHighDensDis', 'Insurant.district', 'district.HighDensDistr')
create dimensional concept InHighDensDis for Insurant AT Insurant.insurant .. Insurant.district AS expand Insurant.byECHighDensDis	INSERT INTO dc_hierarchyexpansion (dcID, tobeexpanded_dcID) VALUES ('Insurant.InHighDensDis', 'Insurant.byECHighDensDis')
create dimensional concept InsInHighDensDis FOR Insurant AT Insurant.insurant AS Insurant.InHighDensDis Insurant.insurant	INSERT INTO dc_conceptLevel (dcID, slice_dcID, levelID) VALUES ('Insurant.InsInHighDensDis', 'Insurant.InHighDensDis', 'Insurant.insurant')
create dimensional concept notHighDensity for Insurant at Insurant.district as (NOT Insurant.byECHighDensDis)	INSERT INTO dc_complement (dcID, negated_dcID) VALUES ('Insurant.notHighDensity', 'Insurant.byECHighDensDis')
create dimensional concept YoungInsInHighDens for Insurant at Insurant.insurant as (Insurant.byECYoungInsurant and Insurant.InsInHighDensDis)	INSERT INTO dc_conjunctive (dcID) VALUES ('Insurant.YoungInsInHighDens') INSERT INTO dc_conjunctive_term (dcID, term_dcID) VALUES ('Insurant.YoungInsInHighDens', 'Insurant.InsInHighDensDis') INSERT INTO dc_conjunctive_term (dcID, term_dcID) VALUES ('Insurant.YoungInsInHighDens', 'Insurant.byECYoungInsurant')
create dimensional concept YoungInsOrHighDens for Insurant at Insurant.insurant as (Insurant.byECYoungInsurant or Insurant.InsInHighDensDis)	INSERT INTO dc_disjunctive (dcID) VALUES ('Insurant.YoungInsOrHighDens') INSERT INTO dc_disjunctive_term (dcID, term_dcID) VALUES ('Insurant.YoungInsOrHighDens', 'Insurant.InsInHighDensDis') INSERT INTO dc_disjunctive_term (dcID, term_dcID) VALUES ('Insurant.YoungInsOrHighDens', 'Insurant.byECYoungInsurant')

Table 6: MDO Syntax and DML statements for example dimensional concepts

[6]For all issued commands see appendix A

3.5. Multi-dimensional Concepts

Multi-dimensional concepts are similar to dimensional and entity concepts. Table 7 shows the MDO syntax and the aspired DML statements for multi-dimensional concepts.

MDO Syntax	**DML statements**
CREATE MULTIDIMENSIONAL FLAT/HIERARCHIC CONCEPT *mdc* FOR *ds* AS	INSERT INTO mdc_mdconcept (mdconceptName, signature_discriminator, definition_discriminator) VALUES (*mdc*,*signature*,*type*) INSERT INTO mdc_flat/hierarchicSignature (mdcId, dimspaceId) VALUES (*mdc*, *ds*)
PRIMITIVE	INSERT INTO mdc_primitive (mdcId) VALUES (*mdc*)
SQL: *query*	INSERT INTO mdc_sqldefined(mdcId, sqlquery) VALUES (*mdc*, *query*)
->(*dr*:*dc*)	INSERT INTO mdc_bydimconcept (mdcid, dimroleid, dcid) VALUES (*mdc*, *dr*,*dc*)
EXPAND mdc'	INSERT INTO mdc_hierarchyexpansion (mdcID, tobeexpanded_mdcID) VALUES (*mdc*,mdc')
$mdc'[ds]$	INSERT INTO mdc_conceptGranularityMDC (mdcID, slice_mdcID, dimspaceID) VALUES (*mdc*, mdc', *ds*)
(mdc_1 AND ... AND mdc_n)	INSERT INTO mdc_conjunctive (mdcId) VALUES (*mdc*) INSERT INTO mdc_conjunctive_term (mdcId, term_mdcID) VALUES (*mdc*,mdc_1) ... INSERT INTO mdc_conjunctive_term (mdcId, term_mdcID) VALUES (*mdc*,mdc_n)
(mdc_1 OR ... OR mdc_n)	INSERT INTO mdc_disjunctive (mdcId) VALUES (*mdc*) INSERT INTO mdc_disjunctive_term (mdcId, term_mdcID) VALUES (*mdc*,mdc_1) ... INSERT INTO mdc_disjunctive_term (mdcId, term_mdcID) VALUES (*mdc*,mdc_n)
(NOT mdc')	INSERT INTO mdc_complement (mdcId, negated_mdcID) VALUES (*mdc*,mdc')

Table 7: Concrete MDO Syntax and resulting SQL inserts into the MDO DB

As with entity concepts and dimensional concepts, the type of the multi-dimensional concept does not need to be stated explicitly, if the user enters a concept definition in a valid form. As described in section 2.3.3 multi-dimensional concepts have, like dimensional concepts, a signature that specifies the granularity of the concept. In contrast to dimensional concepts the user does not specify specific levels for the granularity but only the name of the dimension space. For the parser to be able to distinguish between flat and hierarchical concepts the user has to state the type of concept explicitly when defining a concept as can be seen in table 7, the concepts are all defined using either the keyword 'flat' or 'hierarchic'. Table 8 shows the inserts for our example concepts. To keep the table compact we only show one concept of every concept type and only the insert into the concept specific table and not the insert into the base table.

MDO Syntax	Generated SQL(MDO DB)
create multidimensional flat concept byDcHighDens FOR Ins_Distr AS - >(insurant:Insurant.byECHighDensDis)	INSERT INTO mdc_bydimconcept (mdcid, dimroleid, dcid) VALUES ('byDcHighDens', 'insurant', 'Insurant.byECHighDensDis')
create multidimensional hierarchic concept expandDocInsHPC FOR Ins_Act AS expand DocInsInHPC	INSERT INTO mdc_hierarchyexpansion (mdcID, tobeexpanded_mdcID) VALUES ('expandDocInsHPC', 'DocInsInHPC')
create multidimensional flat concept restrDocInsHPC FOR Ins_Act_persons AS expandDocInsHPC [Ins_Act_persons]	INSERT INTO mdc_conceptGranularityMDC (mdcID, slice_mdcID, dimspaceID) VALUES ('restrDocInsHPC', 'expandDocInsHPC', 'Ins_Act_persons')
create multidimensional flat concept notHighDens FOR Ins_Distr AS (not byDcHighDens)	INSERT INTO mdc_complement (mdcID, negated_mdcID) VALUES ('notHighDens', 'byDcHighDens')
create multidimensional flat concept DocInsInHPC FOR DocIns_Distr AS (byDcDocHPC and byDcInsHPC)	INSERT INTO mdc_conjunctive (mdcID) VALUES VALUES ('DocInsInHPC') INSERT INTO mdc_conjunctive_term (mdcID, term_mdcID) VALUES ('DocInsInHPC', 'byDcInsHPC') INSERT INTO mdc_conjunctive_term (mdcID, term_mdcID) VALUES ('DocInsInHPC', 'byDcDocHPC')
create multidimensional flat concept YoungOrHighDensInsurants FOR Ins_Insurants AS (byDcYoungIns or restHighDensInsurants)	INSERT INTO mdc_disjunctive (mdcID) VALUES ('YoungInsurantsInHighDensDis') INSERT INTO mdc_disjunctive_term (mdcID, term_mdcID) VALUES ('YoungInsurantsInHighDensDis', 'restHighDensInsurants') INSERT INTO mdc_disjunctive_term (mdcID, term_mdcID) VALUES ('YoungInsurantsInHighDensDis', 'byDcYoungIns')

Table 8: MDO Syntax and DML statement examples

3.6. Prototype Implementation and Performance

ANTLR allows us to create the parsers and lexers for a Java implementation when supplied with a correct grammar. For the Java implementation we used ANTLRWorks2 from Tunnelvision[7]. The complete grammar with all lexer and parser rules defined can be found in the Appendix B. In our implementation every kind of concept belongs to a most specific parser rule. ANTLR has two main ways to access rule content, first via the listener pattern and second via visitors classes. ANTLR creates two listener methods for every parser rule, the first method fires when entering the rule and the second method fires when leaving the rule. These methods are created in a base listener class with the default implementation doing nothing. To use the listener methods the base listener class has to be overridden. Sample base listener methods can be seen in listing 12. As

[7]see `http://tunnelvisionlabs.com/products/demo/antlrworks`

we can see every rule has two listener methods, one triggered upon entering a rule and another triggered upon leaving the rule.

```
@Override public void enterDl_create_entityConcept(@NotNull MdoParser.
    Dl_create_entityConceptContext ctx) { }
@Override public void exitDl_create_entityConcept(@NotNull MdoParser.
    Dl_create_entityConceptContext ctx) { }
```

Listing 12: ANTLR listener methods from the MDO base listener

The context in the listener methods in listing 12 called entityConceptContext ctx, can be used to access the values of terminal nodes. Another method for accessing the values of the parse tree is by using the visitor pattern. When we use the visitor pattern we have to manually walk through the parse tree, because of the convenience of the listener methods we chose to implement the parser using the listener pattern.

Figure 14 shows the points where the listener patterns are triggered. The green colored markers show the points where the parser enters a rule, orange shows when it reaches a terminal node, and red shows where it leaves the rule again, as we can see the parse tree is traversed in-order.

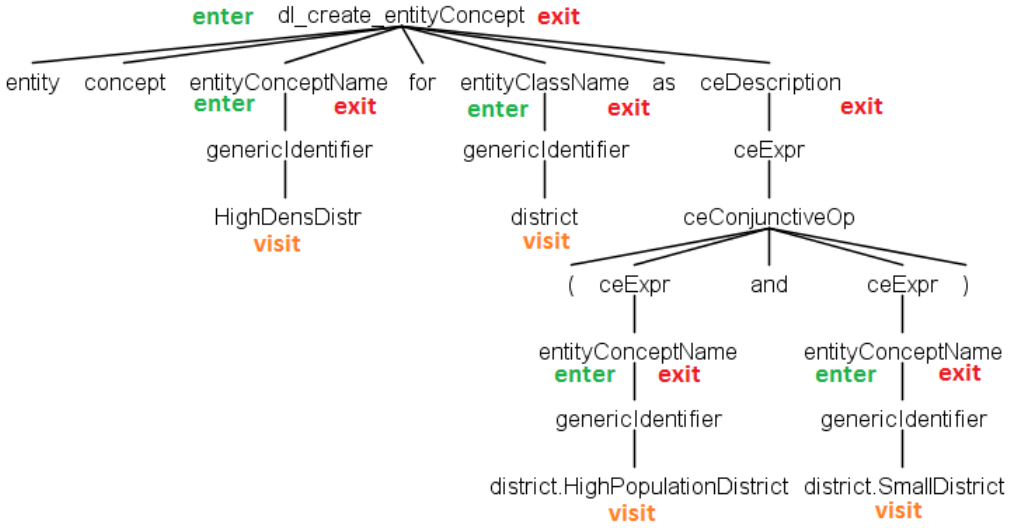

Figure 14: Parsetree with method triggering annotation

In the implementation the MDO parser reads the commands line by line. When entering a command (e.g. creating an entity concept) a parse tree is generated and traversed. The parser saves the values gathered from traversing the resulting parse tree in a Hashmap. These values are upon exit of a parse tree used to create the SQL statements that in turn create the concepts in the MDO-Database. For the Hashmap to know with which key it should save the value inside the map the parser uses flags to show from which part of the parse tree the values are taken. This is necessary because depending on the rule the parser is in, the value of a node has a different meaning. For example a rule that recognizes an entity concept name treats a concept name in different

ways depending on the context e.g. the entity concept name on creating a concept needs to be treated differently than an entity concept name which is the term of a conjunctive entity concept. This problem can be seen when looking at figure 14. In this figure the parse tree enters the rule 'entityConceptName' three times, one time as the name of the conjunctive entity concept that will be created and two times as the entity concept names for the terms.

We will now go step by step through the creation of a conjunctive entity concept (shown in listing 13) to show how the program works. Note that this is only a simplified example[8]. For a better understanding we also commented the code inside the listing. First, when entering a new rule we tell our Hashmap where we are, in our code this is represented by the 'parserState.flags.add' (line 8) method. In this code example the flag 'IN_EC_DEFINITION' is set to indicate that the user wants to define a new entity concept. Upon exit of a parser rule the flag needs to be removed again (line 31). When a terminal node is reached (line 12), the terminal node value is saved for further processing, in this example we save a value by using the 'parserState.ctx.put' method (line 20). In the code we can see that depending on what flag is set (line 15) we either save the value as 'entityConceptName', or add it to a conjunctive term list.

To get a better understanding when the parser calls a certain method we look at figure 14 and listing 13 to describe when the individual methods are called. First the method 'enterdl_create_entityConcept' is called (line 6). After that the rule 'enterEntityConceptName' would be called but the parser does not need to do anything when entering an entity concept name. After entering, the terminal node for the entity concept name is reached which calls the visitTerminal method (line 12). When leaving the entity concept name the method 'exitEntityConceptName' is called (line 14) which saves the entity concept name in the Hashmap.

At the end of a concept definition the corresponding SQL-statements are created and saved for later processing. The SQL-statements are created using the 'SqlGenerator.createEcBinaryConjTerm' method (line 33). The code for creating SQL statements is shown in listing 14 The SqlGenerator class in this example takes the values of the terminal nodes and creates with these values the SQL-statements for the MDO-DB. Listing 14 shows the SqlGenerator methods for creating a conjunctive entity concept. The result of the parsing process is a list of SQL-statements that are issued to the MDO-DB when the parse process is completed.

[8]The whole source code can be found on the DVD enclosed to this thesis

```
1  public class MdoSaxParser extends MdoBaseListener {
2   public void enterMdo_dl(MdoParser.Mdo_dlContext ctx) {
3          //create Object with HashMap and Flags
4          parserState = new ParserState();
5  }
6  public void enterDl_create_entityConcept(MdoParser.
       Dl_create_entityConceptContext ctx){
7          //set Flag that we create an Entity Concept
8          parserState.flags.add(ParserState.FlagsEnum.IN_EC_DEFINITION);
9  };
10 public void visitTerminal(TerminalNode node) {
11                 //PUSH Value of Terminal nodes on Terminal Stack
12                 parserState.pushTerminalStack(node.getText().replace("'", "")
                       );
13 }
14 public void exitEntityConceptName(MdoParser.EntityConceptNameContext ctx){
15         if (parserState.flags.contains(ParserState.FlagsEnum.
               IN_EC_CONJUNCTIVE)){
16                 //save conceptname in termList
17                 termList.add(parserState.popTerminalStack());
18         } else{ //if conceptName is name of the new concept
19                 //save conceptname to hashmap
20                 parserState.ctx.put("entityConceptName", parserState.
                       popTerminalStack());
21         }
22 }
23
24 public void exitEntityClassName(MdoParser.EntityClassNameContext ctx) {
25
26         parserState.ctx.put("entityClassName", parserState.popTerminalStack()
               );
27         }
28
29 public void exitEcConjunctive(@NotNull MdoParser.EcConjunctiveContext ctx) {
30
31         parserState.flags.remove(ParserState.FlagsEnum.IN_EC_CONJUNCTIVE);
32         parserState.parsingResult.add(
33                         SqlGenerator.createEcBinaryConj(
34                         (String)parserState.ctx.get("entityConceptName")
35                         )
36         );
37         while(!termList.empty()){
38                 parserState.parsingResult.add(
39                                 SqlGenerator.createEcBinaryConjTerm(
40                                         (String)parserState.ctx.get("
                                               entityConceptName"),
41                                         termList.pop()
42 ));
43                 }
44         }
45 }
```

Listing 13: Code Snippet of the ANTLR Syntax Parser

```
public static SqlStatement createEcBinaryConj(String name) {
        SqlStatementFactory sqlFac = new SqlStatementFactory();

        sqlFac.sql =
        "INSERT␣INTO␣ec_conjunctive␣(ecId)␣\n" +
        "␣␣VALUES␣(:ecId)";

        sqlFac.sbps.add(new SubstitutionPair(":ecId", name,
                        SubstitutionPair.SQLDatatype.STRING));

        return sqlFac.toSQLStatement();
}

public static SqlStatement createEcBinaryConjTerm(String name, String
    term) {
        SqlStatementFactory sqlFac = new SqlStatementFactory();

        sqlFac.sql =
        "INSERT␣INTO␣ec_conjunctive_term␣(ecId,␣term_ecID)␣\n" +
        "␣␣VALUES␣(:ecId,␣:term)";

        sqlFac.sbps.add(new SubstitutionPair(":ecId", name,
                        SubstitutionPair.SQLDatatype.STRING));

        sqlFac.sbps.add(new SubstitutionPair(":term", term,
                        SubstitutionPair.SQLDatatype.STRING));

        return sqlFac.toSQLStatement();
}
```

Listing 14: Example method for creating SQL statements

Ambiguous Grammars Here we want to shortly show a particular problem we encountered that is in our opinion rather hard to find. The problem occurs when rules are ambiguously defined. An example for an ambiguous definition is listing 15.

```
COMPARISON_OP:  '<' | '>=' | '>' |'=<' | EQUALS_FRG;

LPAREN_ANGLE:   '<';
RPAREN_ANGLE:   '>';
```

Listing 15: Ambiguous Rule Definition

In listing 15 we see that the signs for greater(>) and smaller(<) appear in more than one rule. If the lexer encounters a '<' it does not know if he should tokenize this '<' as a 'LPAREN_ANGLE' or as a 'COMPARISON_OP'. ANTLR allows such grammars. If it cannot determine the right alternative, ANTLR handles this case by choosing the first alternative available (see [Parr, 2013, p.15]) . If the developer is not previously aware of this behavior this leads to errors in the grammar which are hard to find.

Performance Here we want to measure how long it takes when we inserted a number of concepts via our MDO Parser into the database. When we insert a statement via the ANTLR interface, the prototype does three things. First, the transformation of different commands into MDO-DB insert statements. Second, the execution of these statements into the MDO-DB. Third, the mapping of these MDO-Concepts into OWL and DWH statements. The reason for measuring all these three things at once is because of the mapping architecture of the prototype. When a DML statement is created it automatically triggers the mapping execution. Figure 15 shows the measurement results.

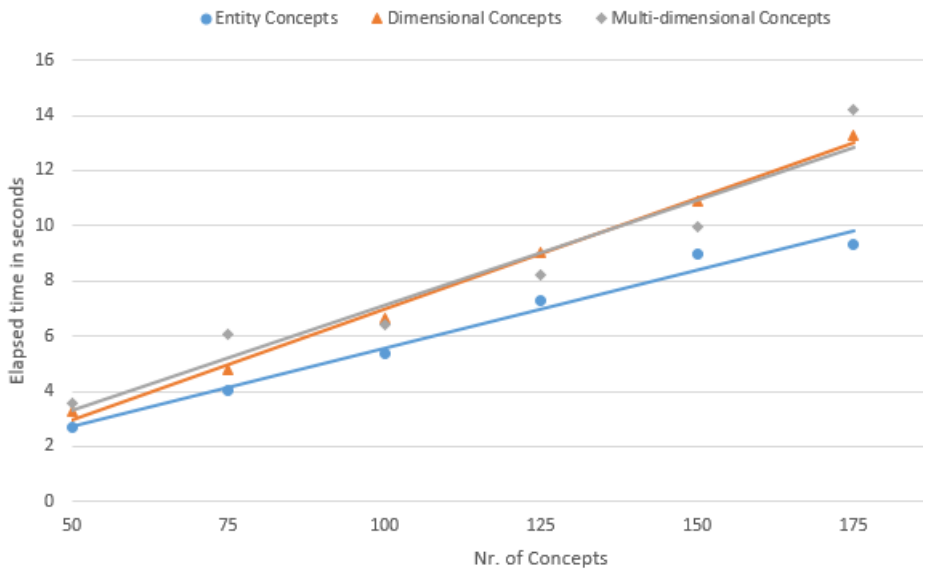

Figure 15: Results for parsing MDO Sytnax statements and executing the created DML statements

The performance measurements show that all different concept types have the same linear behavior. We can see a little difference between entity concepts and the other concept types, this may be because dimensional concepts and multi-dimensional concepts have longer commands and therefore the parser needs to access more rules while processing these concepts. What we can also see is that the process has a satisfactory speed for hand insertion of concepts.

3.7. Discussion

ANTLR is a good framework for mapping MDO syntax into a SQL statements. It is straightforward to write the grammar and the automatic creation of listener methods makes it easy to implement the language into a specific application. Also it is conveniently expanded by updating the grammar and then adding the new listener methods to the existing project. With the use of ANTLR we got rid of the cumbersome and error prone task to write a parser by hand which would most probably have had a worse performance and would not be as easy to implement.

With the implementation of the concrete MDO syntax we are able to give the user a simple user interface to interact with the research prototype. All our test cases where already written with the user interface and this eased the task of concept creation tremendously, especially when creating multi table concepts. For example it takes at least five insert statements to create a dimensional conjunctive concept. One statement for the base table, one for the signature table, one for the context specific dc_conjunctive table and two for the dc_conjunctive term tables. One line of MDO syntax replaces these five statements now.

The performance of the ANTLR inserts is better than expected, which means that our first assumption, that creating a whole OLAP cube from scratch via the interface was not suitable, was wrong. For future projects we should expand the OLAP creation capabilities of the MDO parser.

4. MDO Reasoner: OWL-based reasoning over multidimensional ontologies

To create a shared vocabulary of business terms [Neumayr et al., 2013] identified OWL as suitable solution. OWL has built in reasoning support and satisfiability checks and offers a broad source of API interfaces to work with. The problem with OWL is that it is hard to map the multidimensionality of data warehouses to OWL. A solution for this problem was described by [Neumayr et al., 2013], which is the basis for our mapping implementation in this section.

4.1. Overview

The MDO reasoner consists of two parts, one part is the mapping of concepts to OWL which happens inside the MDO DB, the second part is the reasoning process which happens inside a Java application. The reasoning process is delegated to an external reasoning component, the Hermit reasoner[9][Glimm et al., 2014]. To be able to work with the Hermit reasoner the MDO reasoner uses the OWL API[10][Horridge and Bechhofer, 2011]. Hermit is implemented in Java and uses the OWL 2 DL[Hitzler et al., 2012] version of OWL. The Hermit reasoner processes OWL files to create a subsumption hierarchy for our concepts. We chose to create the OWL files in Manchester Syntax[Horridge and Patel-Schneider, 2009] to make them human readable. To create the OWL file we first need a mapping from our MDO-DB concepts to OWL. The base for the mapping between the MDO and OWL is the paper from [Neumayr et al., 2013]. In the implementation we chose to implement the OWL mapping via trigger so the user can instantly inspect the mapping results of an added concept.

To make the reasoning process clearer we look at figure 16. First we have a command for creating a new entity concept. This command is processed by the MDO parser to create insert statements for the MDO DB. When the statements are inserted in the MDO DB the mapping process for the inserted concepts is started. The OWL axioms for the mapped concepts are also stored inside the MDO DB. To infer over the concept definitions our MDO reasoner collects all OWL axioms inside the MDO and creates an OWL file which is then transferred to the Hermit reasoner for creating the subsumption hierarchy. The MDO reasoner receives the subsumption hierarchy from the Hermit reasoner and persists it inside the MDO DB.

Before we discuss how we implemented the mapping we first give a short introduction to reasoning and look at how to map our concepts. The reasoner is a component that allows the program to create a so called subsumption hierarchy over business terms, our concepts. The terms in this hierarchy are ordered from general terms to more specific terms. For example, if we have the two terms 'Big District', defined as a district having more than 1 Million inhabitants, and 'Huge District', defined as district having more than

[9] http://hermit-reasoner.com/java.html
[10] http://owlapi.sourceforge.net/

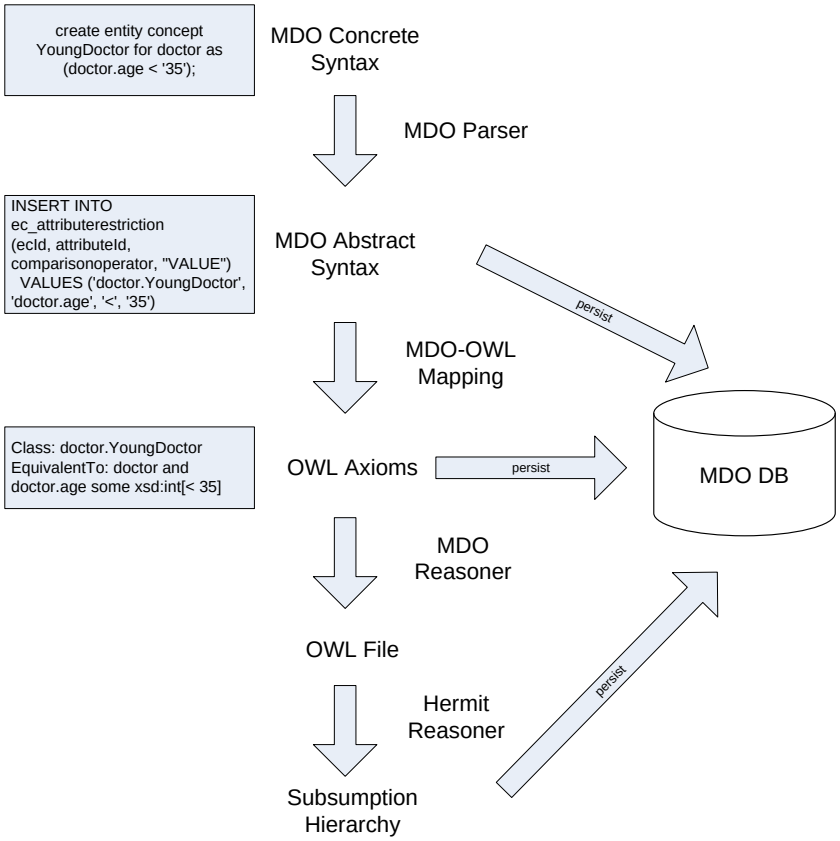

Figure 16: Reasoning process from user input to subsumption hierarchy

10 Million inhabitants, the reasoner can determine that 'Huge District' is a more special term than 'Big District'. Therefore 'Huge District' is in the subsumption hierarchy below 'Big District'. In other words, the reasoner can tell the user the following: Every 'Huge District' is also a 'Big District' but not every 'Big District' is a 'Huge District'. With this subsumption hierarchy we can bring order into our multitude of concepts which makes the work with concepts much more convenient. The result of such a reasoning process can be seen in figure 3 at the beginning of this thesis.

This section is structured as followed, first we will look how we map our concepts into OWL syntax, than we show how we implemented the mapping procedure and last we will look how our Java program of the reasoning process is implemented.

4.2. OLAP Cube Representation

For the mapping from MDO to OWL to work we need an OWL representation of an OLAP cube. This representation is taken from [Neumayr et al., 2013, p.8] and shown in Table 9. This table contains the concepts for representing the instances of an OLAP Cube in Manchester Syntax.

	OWL (Manchester Syntax)
(1)	Class: **Entity**
(2)	Class: **Node** SubClassOf: directlyRollsUpTo Self and roleOf some Entity and atLevel some Level and rollsUpTo only Node
(3)	Class: **Level** SubClassOf: directlyRollsUpTo Self and rollsUpTo only Level
(4)	Class: **Point**
(5)	DisjointClasses: Entity, Node, Level, Point
(6)	ObjectProperty: **directlyRollsUpTo** SubPropertyOf: rollsUpTo
(7)	ObjectProperty: **rollsUpTo** Characteristics: Transitive
(8)	ObjectProperty: **roleOf** Characteristics: Functional Domain: Node Range: Entity
(9)	ObjectProperty: **atLevel** Characteristics: Functional Domain: Node Range: Level
(10)	for the set of all entities $\{e_1, \ldots, e_n\}$: DifferentIndividuals: $e_1, \ldots, e_n$
(11)	for the set of all entity classes $\{ecl_1, \ldots, ecl_n\}$: DisjointClasses: $ecl_1, \ldots, ecl_n$
(12)	for the set of all dimensions $\{d_1, \ldots, d_n\}$: DisjointClasses: $d_1, \ldots, d_n$

Table 9: Classes and properties for OLAP cube representation in Manchester Syntax

Table 9 shows the most basic artifacts. As we can see we still miss for example dimensions or dimension spaces. The additional OLAP artifacts will be introduced in the section where they are used first. A complete discussion over the representation of OLAP Cubes in OWL can be found in [Neumayr et al., 2013].

4.3. Entity Concepts

Entity concepts are defined over entityclasses. The mapping of entity classes to OWL and the Manchester Syntax can be found in table 11 and is taken from [Neumayr et al., 2013, p.6]. Because the paper from [Neumayr et al., 2013] showed all mappings of concepts in description logic notation and the implementation of the MDO reasoner uses Manchester Syntax for representing concepts in OWL we chose to show both notations in the mapping tables throughout this section.

An entity class consists of a name, and has a number of attributes. Every attribute has a data type asserted. All entities in an OLAP cube belong to a certain entity class. Entity classes are also the basic building block for all entity concepts, every entity concept is defined for exactly one entity class. So entities belong to one entity class and can belong to many entity concepts, depending on their attribute values.

OWL (DL notation)	**OWL (Manchester Syntax)**
$ecl \sqsubseteq$ Entity $\exists attr_1.\top \sqsubseteq ecl \ldots \exists attr_n.\top \sqsubseteq ecl$ $\top \sqsubseteq \forall attr_1.dt \ldots \top \sqsubseteq \forall attr_n.dt$ $\top \sqsubseteq \leqslant 1 attr_1 \ldots \top \sqsubseteq \leqslant 1 attr_n$	Class: *ecl* SubclassOf: Entity DataProperty: $attr_1$ Characteristics: Functional Domain: *ecl* Datatype: dt_1 ... DataProperty: $attr_n$ Characteristics: Functional Domain: *ecl* Datatype: dt_n

Table 11: Entity class OWL representation in DL notation and Manchester Syntax [Neumayr et al., 2013, p.6]

As already introduced in section 2.3.1 there are seven different supported entity concepts: Primitive, SQL-defined, Nominal, Attribute Restricted, Conjuntive, Disjunctive and Complement. The corresponding OWL mapping from the MDO for the different concepts is shown in table 12 and is taken from [Neumayr et al., 2013, p.10]. We are not able to reason over primitive or sql-defined concepts, this is represented by a blank colum in table 12, because they are defined without the usage of MDO constructs, but we could use these concepts in combination with other concepts for example conjunctive concepts. As table 12 shows the representation of entity concepts in OWL is straightforward.

MDO Syntax	**OWL (DL notation)**	**OWL (Manchester Syntax)**
CREATE ENTITY CONCEPT *ec* FOR *ecl*	$ec \sqsubseteq ecl$	Class: ec SubclassOf: ecl
AS PRIMITIVE; AS SQL *sqlquery*;		
AS	$ec \equiv$	EquivalentTo:
$entity_1, \ldots, entity_n$;	$\{entity_1, \ldots, entity_n\}$	$(\{entity_1, \ldots, entity_n\})$
attr θ *value*;	$\exists attr.dt[\theta\ value]$	attr SOME $dt[\ \theta\ value]$
(ec_1 OR ... OR ec_n);	$ec_1 \sqcup \cdots \sqcup ec_n$	(ec_1 OR ... OR ec_n)
(ec_1 AND ... AND ec_n);	$ec_1 \sqcap \cdots \sqcap ec_n$	(ec_1 AND ... AND ec_n)
NOT ec';	$ecl \sqcap \neg ec$	ecl and NOT (ec')

Table 12: Representation of entity concepts in OWL

Table 13 shows how the example concepts are mapped to OWL concept definitions. The result of the MDO OWL mapping is stored in table entityconcepts_inowl in the MDO-DB.

With the concepts mapped to OWL axioms the Hermit reasoner can now infer the subsumption hierarchy over the test concepts. The resulting subsumption hierarchy of the reasoning process is shown in figure 17.

The reasoner detects that VeryYoungDoctor(defined as having an age lower 30 years) is a subconcept of YoungDoctor, that HighDensityDistrict is subconcept of HighPopulation and SmallDistrict and that, OlldDoctor and YoungDoctor are both subconcepts of

MDO Syntax	Manchester Syntax
create entity concept myProvince for province as province.Niederösterreich	Class: province.myProvince EquivalentTo: province and ({province.Niederösterreich})
create entity concept YoungDoctor for doctor as (doctor.age <'35')	Class: doctor.YoungDoctor EquivalentTo: doctor and doctor.age some xsd:int[<35]
create entity concept HighDensDistr for district as (district.HighPopulationDistrict and district.SmallDistrict)	Class: district.HighDensDistr EquivalentTo: district and (district.HighPopulationDistrict and district.SmallDistrict)
create entity concept YoungOrOldDoc for doctor as (doctor.OldDoctor or doctor.YoungDoctor)	Class: doctor.YoungOrOldDoc EquivalentTo: doctor and (doctor.OldDoctor or doctor.YoungDoctor)
create entity concept notSmallDistrict for district as (NOT district.SmallDistrict)	Class: district.notSmallDistrict EquivalentTo: district and not(district.SmallDistrict)

Table 13: Manchester syntax representation of example entity concepts

OldOrYoungDoctor. In figure 17, Protégé[11] is used to visualize the inferred subusmption hierarchy. The resulting subsumption hierarchy is stored in the subsumption table ec_subsumption in the MDO-DB.

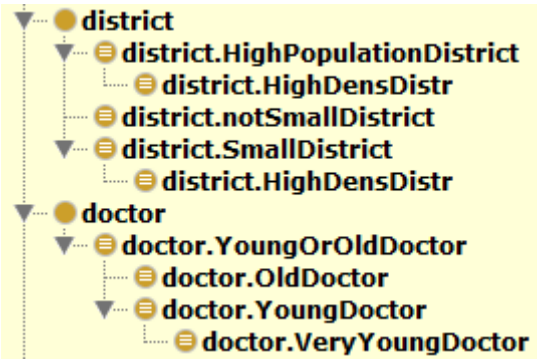

Figure 17: Subumption hierarchy of example entity concepts

[11] http://protege.stanford.edu/

4.4. Dimensional Concepts

As mentioned in the beginning, dimensional concepts have a dimension as their domain and consist of a subset of nodes of this dimension. For the representation of dimensional concepts we first need a representation of the OLAP concepts of dimensions. These concepts and OWL mappings are shown in table 15.

MDO Syntax	**OWL (DL notation)**	**OWL (Manchester Syntax)**
CREATE DIMENSION d WITH LEVELS l_1 ecl_1, ..., l_n ecl_n AND HIERARCHY l'_1 UNDER l''_1, ..., l'_m UNDER l''_m;	$d \sqsubseteq$ Node $\exists$atLevel.$\{l_1\} \equiv d \sqcap \exists$roleOf.$ecl_1$ ... $\exists$atLevel.$\{l_n\} \equiv d \sqcap \exists$roleOf.$ecl_n$ directlyRollsUpTo(l'_1, l''_1) ... directlyRollsUpTo(l'_m, l''_m)	Class: d SubClassOf: Node (atLevel some l_1) EquivalentTo (d and roleOf some ecl_1) ... (atLevel some l_n) EquivalentTo: (d and roleOf some ecl_n) Individual:l'_1 directlyRollsUpTo l''_1 ... Individual:l'_n directlyRollsUpTo l''_n
CREATE LEVELRANGERESTRICTED DIMENSION lrd AS $d[l'..l'']$;	$lrd \equiv d \sqcap \exists$atLevel.($\exists$rollsUpTo$^-$.$\{l'\} \sqcap$ $\exists$rollsUpTo.$\{l''\}$)	Class: lrd EquivalentTo: Level and (inverse(rollsUpTo value l')) and (rollsUpTo value l'')

Table 15: OLAP Dimensions in OWL

Level-range-restricted dimensions In [Neumayr et al., 2013] dimensional concepts are defined over so called level-range-restricted dimensions, these dimensions and their corresponding nodes are defined over a top level and a bottom level of a dimension. Every level-range is indirectly restricted to a certain dimension as every level is asserted to only one dimension. In our implementation we have two slight differences to this approach, first, we have the special case of flat dimension where the dimension consists of only one level. This case can be seen as a level-range-restriction where top and bottom level are the same.

Second, in our implementation we do not have the concept of level-range-restricted dimensions represented, dimensional concepts are not defined over level-range-restricted dimensions but over whole dimensions. This domain(the dimension) is then restricted by the signature of the concept, the signature in the implementation acts as the level-range-restriction of the dimension. When a dimensional concept with a new signature is created we create a level-range-restriction accordingly. The level-range-restrictions are created with the following naming pattern: $dimensionname$+'Fr'+$fromLevel$+'To'+$toLevel$.

For example a restricted level range from the insurant dimension could be called 'InsurantFrDistrictToProvince'. When the user enters many concepts it can occur that two concepts have the same level range and therefore the same name for a level-range-restriction. We do not check for duplicates, we just insert them in our level-range

table. To prevent duplicates from causing errors in the database the IGNORE_ROW-_ON_DUPKEY_INDEX[12] hint was used. When a duplicate key causes a primary key violation, the insert into the table is simply discarded.

Level Disjointness [Neumayr et al., 2013] find in their paper a key problem with representing dimensions in OWL, the disjointness of sub-dimensions. The problem occurs because of the inability of OWL to define transitive properties as functional. We would need this functionality to represent roll-up-hierarchies. The characteristic of such hierarchies is that they are transitive and one node of a lower level always rolls up to exactly one node of a higher level. The problem of not recognized sub-dimensions is defined by [Neumayr et al., 2013, p.13] as follows: ' it[the subsumption hierarchy] will not recognize that if two concepts dc and dc' are disjoint, their hierarchical expansions dc* and dc'*, are disjoint too'.

To solve this problem [Neumayr et al., 2013] proposed to introduce redundant information for levels and nodes. For every level they introduced a functional rollup property 'rollup_levelname'. Every descendant of a named node of a particular level rolls up to this node using the 'rollup_levelname' property. The OWL representation of this behavior is shown in table 17.

OWL (DL notation)	**OWL (Manchester Syntax)**
for each level l:	
$\exists$atLevel.$\exists$rollsUpTo.$\{l\} \equiv \exists$rollsUpTo_$l.\top$	(atLevel some (rollsUpTo some l) EquivalentTo: (rollsUpTo_$l.\top$)
rollsUpTo_$l \sqsubseteq$ rollsUpTo	ObjectProperty: rollsUpTo_l SubPropertyOf: rollsUpTo
$\top \sqsubseteq \forall$rollsUpTo_$l.\exists atLevel.\{l\}$	Range: (atLevel some l)
$\top \sqsubseteq \leqslant 1$rollsUpTo_$l$	Characteristics: Functional
for each node nd at level l:	
$\exists$rollsUpTo.$\{nd\} \equiv \exists$rollsUpTo_$l.\{nd\}$	(rollsUpTo some nd) EquivalentTo: (rollsUpTo_l some nd)

Table 17: Representing disjointness of levels in OWL [Neumayr et al., 2013, p.12]

After all necessary parts are defined for representing dimensional concepts in OWL the mapping of dimensional concepts is shown in table 18. The corresponding mapping of our use case concepts is shown in table 19.

Figure 18 shows the reasoning result of the defined concepts and they are as expected. We only showed the subsumption hierarchy for the concepts of the insurant dimension as we only defined one concept on the doctor dimension and this concept cannot be reasoned because it is an SQL defined concept.

To test if the proposed solution for representing disjointness of levels works we checked if the expansions of 'byECHighDensDis' and 'NotHighDensity' are being recognized as

[12]For further information see `http://docs.oracle.com/cd/E11882_01/server.112/e41084/sql_elements006.htm#SQLRF30052`

MDO Syntax	**OWL (DL notation)**	**OWL (Manchester Syntax)**
CREATE DIMENSIONAL CONCEPT dc FOR d AT lr	$dc \sqsubseteq d \sqcap \exists atLevel.lr$	Class: dc SubclassOf: d and atLevel some lr
AS PRIMITIVE; AS SQL *sqlquery*;		
AS	$dc \equiv d \sqcap \exists atLevel.lr \sqcap$	EquivalentTo: d AND atLevel some lr
->$level$: ec;	$\exists atLevel.level \sqcap$ $\exists$roleOf.ec	atLevel some $level$ AND roleOf some ec
EXPAND dc';	$\exists$rollsUpTo_l.dc'	rollsUpTo_l some dc'
dc' lr;	$dc' \sqcap lr$;	dc' AND atLevel value lr
dc_1 OR ... OR dc_n);	$dc_1 \sqcup \cdots \sqcup dc_n$	(dc_1 OR ... OR dc_n)
(dc_1 AND ... AND dc_n);	$dc_1 \sqcap \cdots \sqcap dc_n$	(dc_1 AND ... AND dc_n)
(NOT dc');	$\neg dc'$	NOT (dc')

Table 18: Dimensional concept mapping

MDO Syntax	**Manchester Syntax**
create dimensional concept SqlMayer for Doctor at Doctor.doctor AS sql 'select doctor from d_doctor where doctor = 'Mayer'	Class: Doctor.SqlMayer SubClassOf: Doctor and atLevel some DoctorFrdoctorTodoctor (SQL-defined concepts not represented in OWL)
create dimensional concept byECHighDensDis for Insurant AT Insurant.district AS ->(Insurant.district:district.HighDensDistr)	Class: Insurant.byECHighDensDis EquivalentTo: Insurant and atLevel some InsurantFrdistrictTodistrict and roleOf some district.HighDensDistr
create dimensional concept InHighDensDis for Insurant AT Insurant.insurant .. Insurant.district AS expand Insurant.byECHighDensDis	Class: Insurant.InHighDensDis EquivalentTo: Insurant and atLevel some InsurantFrinsurantTodistrict and rollsUpTo_district some Insurant.byECHighDensDis
create dimensional concept InsInHighDensDis FOR Insurant AT Insurant.insurant AS Insurant.InHighDensDis Insurant.insurant	Class: Insurant.InsInHighDensDis EquivalentTo: Insurant and atLevel some InsurantFrinsurantToinsurant and Insurant.InHighDensDis and atLevel value Insurant.insurant
create dimensional concept notHighDensity for Insurant at Insurant.district as (NOT Insurant.byECHighDensDis)	Class: Insurant.notHighDensity EquivalentTo: Insurant and atLevel some InsurantFrdistrictTodistrict and not Insurant.byECHighDensDis
create dimensional concept YoungInsInHighDens for Insurant at Insurant.insurant as (Insurant.byECYoungInsurant and Insurant.InsInHighDensDis)	Class: Insurant.YoungInsInHighDens EquivalentTo: Insurant and atLevel some InsurantFrinsurantToinsurant and (Insurant.InsInHighDensDis and Insurant.byECYoungInsurant)
create dimensional concept YoungInsOrHighDens for Insurant at Insurant.insurant as (Insurant.byECYoungInsurant or Insurant.InsInHighDensDis)	Class: Insurant.YoungInsOrHighDens EquivalentTo: Insurant and atLevel some InsurantFrinsurantToinsurant and (Insurant.InsInHighDensDis or Insurant.byECYoungInsurant)

Table 19: Example dimensional concepts and their representation in Manchester Syntax

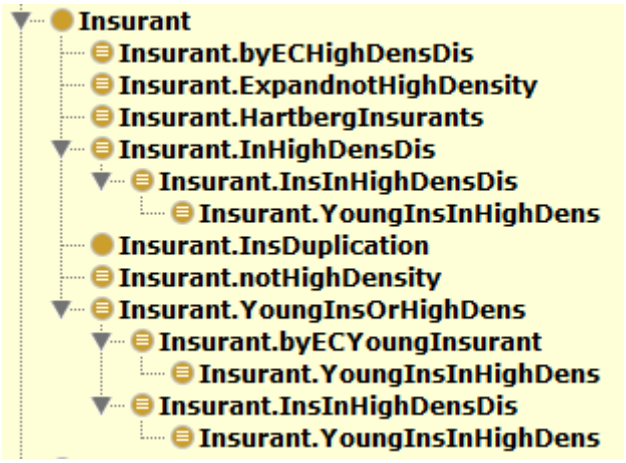

Figure 18: Subsumption hierarchy of dimensional concepts on the insurant dimension

disjoint. The result: They are being recognized as disjoint as figure 19 shows. Unfortunately Protégé does not show the disjoint classes therefore we took a screen shot of our implemented 'disjoint_with' table that holds all disjoint concepts.

	DCID1	DCID2
1	Insurant.ExpandnotHighDensity	Insurant.InHighDensDis
2	Insurant.ExpandnotHighDensity	Insurant.InsInHighDensDis
3	Insurant.ExpandnotHighDensity	Insurant.YoungInsInHighDens

Figure 19: Disjointness of expanded dimensional concepts

4.5. Multi-dimensional Concepts

As a recap, multi-dimensional concepts are defined over dimension spaces. A dimension space consists of a number of dimension roles. The mapping of dimension spaces and dimension roles to OWL can be found in table 21 and the OWL mapping is taken from [Neumayr et al., 2013, p.6].

MDO Syntax	**OWL (DL notation)**	**OWL (Manchester Syntax)**
CREATE DIMENSION SPACE ds AS $(dr_1lr_1, \ldots, dr_nlr_n)$	$ds \equiv \text{Point} \sqcap \exists dr_1.\exists atLevel.lr_1 \sqcap \cdots \sqcap \exists dr_n.\exists atLevel.lr_n$	Class: ds EquivalentTo: Point and (dr_1 atLevel some lr_1) ... and (dr_n atLevel some lr_n)
CREATE DIMENSION ROLE dr OF d;	$\top \sqsubseteq \leqslant 1 dr$ $\exists dr.\top \sqsubseteq \text{Point}$ $\top \sqsubseteq \forall dr.d$	ObjectProperty: dr Characteristics: Functional Domain: Point Range: d

Table 21: OLAP mapping of dimension spaces and dimroles

Table 22 shows the mapping of concepts to OWL in description logic notation and Manchester Syntax.

MDO Syntax	**OWL (DL notation)**	**OWL (Manchester Syntax)**
CREATE MULTIDIMENSIONAL CONCEPT mc FOR ds	$mc \sqsubseteq ds$	Class: mc SubclassOf: ds
AS PRIMITIVE; AS SQL $sqlquery$;		
AS	$mc \equiv$	EquivalentTo:
->(dr:dc);	$\exists dr.dc$	dr some dc
EXPAND mc';	see text	see text
$mc'[ds]$;	$mc' \sqcap ds$	mc' and ds
(mc_1 OR ... OR mc_n);	$mc_1 \sqcup \cdots \sqcup mc_n$	(mc_1 OR ... OR mc_n)
mc_1 AND ... AND mc_n);	$mc_1 \sqcap \cdots \sqcap mc_n$	(mc_1 AND ... AND mc_n)
(NOT mc');	$ds \sqcap \neg mc'$	ds AND NOT (mc')

Table 22: Multi-dimensional concepts mapping

To get a reasoning over hierarchical multi-dimensional concepts the multi-dimensional concepts have to be transformed into disjunctive normal form. In short, for the hierarchy expansion to work we have to brake up the multidimensional concept into its dimensional concepts and the dimension roles they are referred to. Then the dimensional concepts referred by the same dimension role are joined again. For this purpose a recursive pro-

cedure was written[13] which dissects the different dimension roles and applies the correct disjunctive normal form. In our use case we expanded a byDimensional concept. To expand this concept our mapping procedure changed the original concept from referencing the dimensional concept with the operator 'some', to referencing the concept with the object property 'rollsUpTo'. A broader explanation how the disjunctive normal form works and why it is needed is given in [Neumayr et al., 2013, p. 14]. Table 23 shows how our example multi-dimensional concepts are mapped to Manchester Syntax.

MDO Syntax	**Manchester Syntax**
create multidimensional flat concept byDcHighDens FOR Ins_Distr AS ->(insurant:Insurant.byECHighDensDis)	Class: byDcHighDens SubClassOf: Ins_Distr EquivalentTo: dr_insurant some Insurant.byECHighDensDis
create multidimensional hierarchic concept expandDocInsHPC FOR Ins_Act AS expand DocInsInHPC	expandDocInsHPC SubClassOf: Ins_Act EquivalentTo: (((dr_actDoc some (rollsUpTo some Doctor.DocHPC))) and ((dr_insurant some (rollsUpTo some Insurant.InsHPC))))
create multidimensional flat concept restrDocInsHPC FOR Ins_Act_persons AS expandDocInsHPC [Ins_Act_persons]	Class: restrDocInsHPC SubClassOf: Ins_Act_persons EquivalentTo: Ins_Act_persons and expandDocInsHPC
create multidimensional flat concept notHighDens FOR Ins_Distr AS (not byDcHighDens)	Class: notHighDens SubClassOf: Ins_Distr EquivalentTo: Ins_Distr and not(byDcHighDens)
create multidimensional flat concept DocInsInHPC FOR DocIns_Distr AS (byDcDocHPC and byDcInsHPC)	Class: DocInsInHPC SubClassOf: DocIns_Distr EquivalentTo: (byDcDocHPC and byDcInsHPC)
create multidimensional flat concept YoungOrHighDensInsurants FOR Ins_Insurants AS (byDcYoungIns or restHighDensInsurants)	Class: YoungOrHighDensInsurants SubClassOf: Ins_Insurants EquivalentTo: (byDcYoungIns or restHighDensInsurants)

Table 23: Example multi-dimensional concepts

Figure 20 shows the reasoning results as expected. Ins_DisIns, Ins_Distr and Ins_Insurants are the dimension spaces containing the concepts.

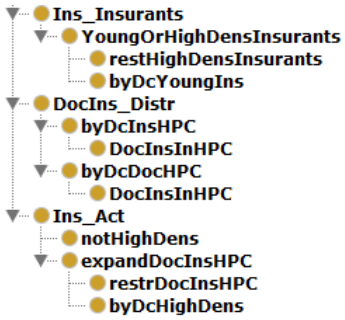

Figure 20: Subsumption hierarchy of multi-dimensional example concepts

[13]The source code for the procedure is on the DVD enclosed to this master thesis

4.6. Prototype Implementation and Performance

In this section we will describe how we implemented the MDO reasoner that consists of two parts: the mapping of MDO concepts that happens inside the MDO DB, and the Java class for interacting with the OWL API and the Hermit Reasoner, called SubsumptionBuilder.

In subsection 4.6.1 we will see how the specific mapping for the different concepts is implemented and see their prerequisites. In subsection 4.6.2 we describe the implementation of the SubsumptionBuilder.

4.6.1. Implementation of the MDO to OWL transformation

One requirement of the reasoning component was that the mapping between the MDO-database and OWL happens instantly; every time the user adds a new concept the user should be able to inspect the resulting OWL mapping in Manchester Syntax[14]. To fulfill this requirement the mapping happens inside the database through triggers that act, if manipulations on the MDO tables, representing the concepts, occur. An example of such a trigger is shown in listing 16. The example shows the trigger for mapping attribute restricted entity concepts to OWL.

```
CREATE OR REPLACE TRIGGER attributeec_inowl_trigger
AFTER INSERT OR UPDATE OR DELETE ON ec_attributerestriction
  FOR EACH ROW
  BEGIN
     IF (DELETING) THEN
       DELETE FROM entityconcept_inOwl WHERE ecID = :OLD.ecID;
      ELSIF (UPDATING) THEN
       UPDATE entityconcept_inOwl
       SET ecID = :NEW.ecID, owl = 'Class:␣' || :NEW.ecID || '␣
           EquivalentTo:␣' || :NEW.entityClassID || '␣and␣' || :NEW.
           attributeID || '␣some␣' || :NEW."VALUE"
       WHERE ecID = :OLD.ecID;
     ELSIF (INSERTING) THEN
       INSERT INTO entityconcept_inOwl
       VALUES (:NEW.ecID,'Class:␣' || :NEW.ecID || '␣EquivalentTo:␣' || :
           NEW.entityClassID || '␣and␣'|| :NEW.attributeID || '␣some␣' ||
             inowl_functions.getComparisonterm(:NEW.comparisonOperator,:
           NEW."VALUE",:NEW.attributeID));
     END IF;
   END;
```

Listing 16: PL\SQL trigger for attribute restricted entity concept mapping

The OWL Axioms for the concepts are created by using string concatenation, if an update on a concept table occurs, the whole OWL Axiom is rewritten.

[14]For an overview over the Syntax see http://www.w3.org/TR/owl2-manchester-syntax/

inOwl Tables The results of the mapping process are stored in inOWL tables which consist of an id, identifying the mapped MDO construct, and the corresponding Manchester Syntax mapping. The structure of a sample inOWL table is shown in listing 17. This example table contains the mapping results for entity concepts.

```
CREATE TABLE entityconcept_inOwl(
  ecID varchar2(201) Primary Key,
  owl varchar2(400)
);
```

Listing 17: inOWL table for storing mapping results

The mapping results are later processed by the external reasoning Java implementation, called SubsumptionBuilder. The reason for using inOWL tables with Manchester Syntax is to make the mappings human readable. The users should be able to verify the mapping result and modify them if they need to.

Compound Trigger Another type of trigger was used to avoid the mutating table problem[15] for multi table concepts. This problem occurs if we want to write into a table and at the same time read from it. To avoid the problem we used compound trigger[16], an example compound trigger can be seen in listing 18.

```
CREATE OR REPLACE TRIGGER nominal_entity_inowl_trigger
    FOR DELETE or UPDATE or INSERT
    ON ec_nominal_entity
    COMPOUND TRIGGER
      TYPE changes IS TABLE OF VARCHAR2(100);
      changeTable changes := changes();
      AFTER EACH ROW IS
      BEGIN
      changeTable.extend;
      changeTable(changeTable.count) := :NEW.ecID;
      END AFTER EACH ROW;
      AFTER STATEMENT IS
      BEGIN
      FOR i IN changeTable.FIRST .. changeTable.LAST
        Loop
          UPDATE entityconcept_inOwl
          SET owl = REGEXP_REPLACE(owl,'EquivalentTo:.*','EquivalentTo:␣'
               || REGEXP_REPLACE(ecID, '(\..*)') || '␣and␣(' ||
              inowl_functions.getnominalEntities(changeTable(i))) || ')'
               where ecID = changeTable(i);
        end loop;
      END AFTER STATEMENT;
  END nominal_entity_inowl_trigger;
```

Listing 18: Compound trigger for mapping nominal entity concepts

[15] A discussion of how to avoid this problem can be found here http://asktom.oracle.com/pls/asktom/ASKTOM.download_file?p_file=6551198119097816936

[16] for a description of compound triggers see http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#LNPLS2005

This trigger works as follows, first the after each row part, captures the inserts or updates that where made on the ec_nominal_entity table and stores them in the variable changeTable. After the statements are executed we can change the mapping result in the inOwl table.

As already described in section 2.3 concepts can be divided into single table and multi table concepts. Next we see how the OWL mapper treats these different concept types.

Single Table Concepts Single table concepts are concepts which consist of only one concept specific table in the MDO-DB. They have in common that the mapping process can happen through the use of standard triggers.

The mapping of these concepts happens as follows: A trigger is assigned to the table we want to map, the trigger than fires every time the user inserts or changes data in this table. The trigger inserts the mapping result into the corresponding inOWL table (in this example enitityconcept_inOWL). The mapping of primitive and sql-defined concepts to OWL is straightforward and is done by string concatenation in the form as shown in listing 19.

```
INSERT INTO entityconcept_inOwl (ecID, OWL)
VALUES (ecID, 'Class:␣' || ecID || '␣SubClassOf:␣' || eclID)
```

Listing 19: Primitive SQL-defined concept mapping procedure

For the attribute-restricted and complement concept the mapping procedure is the same with the only difference that these concepts have some additional columns which have to be mapped according to table 12.

Multi Table Concepts Multi table concepts in contrast to single table concepts consist of more than one concept specific table, so we have to define more triggers (the most common case being two) two map the concepts accordingly. Another characteristic is that most multi table concepts are mapped via compound triggers. As an example we take the nominal entity concept which consists of the ec_nominal table containing the name of the concept and the ec_nominal_entity table containing the entities assigned to the nominal concepts. First with the insertion of the concept in the ec_nominal table the concept is created like a single table concepts but with the special object property 'isEmpty' as shown in listing 20.

```
INSERT INTO entityconcept_inOwl (ecID, OWL)
VALUES (ecID, 'Class:␣' || ecID || '␣SubClassOf:␣' || eclID ||
 '␣EquivalentTo:␣isEmpty')
```

Listing 20: Nominal concept mapping procedure

The object property 'isEmpty' in this case is only a flag to show that the concept does not have entities assigned to it yet and prevents the ontology to be in an inconsistent stage if the user decides to conduct a reasoning with no entities yet assigned.

The next step in the mapping process is the adding of single entities to the according concepts. To ensure that the mapped concept is always in a valid state, the whole 'EquivalentTo' part of the concept is rewritten when a new dataset is inserted or an

existing dataset is altered. As performance was not in the focus of the implementation the repetitive reading of the table was deemed to be better than the error prone approach of interchanging single entities with means of different string operations. Here we used the before introduced compound trigger. The reason why we need a compound trigger is because we want to read the whole ec_nominal_entity table when a new dataset is added or altered and this raises the mutating table exception without compound triggers. The mapping itself is, because of the chosen approach, very straightforward as shown in listing 21.

```
UPDATE entityconcept_inOwl
SET owl = REGEXP_REPLACE(owl,'EquivalentTo:.*','EquivalentTo: ' || eclID
    || ' and (' || inowl_functions.getnominalEntities(ecID) || ')'
WHERE ecID = :NEW.ecID;
```

Listing 21: Nominal entity mapping procedure

The regex operation takes the old EquivalentTo part and replaces it with the new EquivalentTo part. The function getnominalEntities[17] reads all the nominal entities assigned to the nominal entity concept and concatenates them to get a list of entities separated by a comma. The other multi table concepts work according to the nominal entity concepts.

4.6.2. Implementation of the Subsumption Builder

The reasoning part of the implementation was built using Java and the Hermit OWL reasoner. The main components of the Java implementation where: the OWL API, the OWL mapping described before and JDBC. With the OWL API we where able to facilitate all the OWL features and get off-the-shelf reasoning support for our previously created ontologies. The class responsible for orchestrating the different program parts is, as earlier mentioned, called SubsumptionBuilder.

The SubsumptionBuilder works in the following way, first a JDBC connection to the MDO DB is created to read the mapped concepts. Then the SubsumptionBuilder creates an OWL file which is handed over to the Hermit reasoner. The reasoner infers the subsumption hierarchy and then the SubsumptionBuilder persists the subsumption hierarchy inside the MDO DB, the following section describes the individual tasks in more detail.

Database connection The connection with the database is created using JDBC. To read all the mappings we first have to construct a query with a special pattern that collects all mapping results from the different inOWL tables. The query is shown in listing 22. With this query we can read the mapping result using JDBC.

[17]The source code for our utility functions can be found on the DVD enclosed to the master thesis

```
select owl from entityconcept_inowl
UNION
select owl from dimensionalconcept_inowl
UNION ...
```

Listing 22: Fragment of the query to collect all OWL mappings

OWL File Now that we have obtained the mappings, we can start constructing an owl file which can later be used by the OWL API and the Hermit reasoner. For an OWL file to be valid we need to give the ontology a name and define ontology and class prefixes. The names and prefixes need to be in the form of URIs. For this project we chose the URI < http://dke.uni-linz.ac.at/mdo >the URI is only needed for the Hermit reasoner to work, it does not have any other purpose and therefore any name for the URI would suffice. Another prerequisite is that the OWL file is written in a supported syntax; we chose to map our MDO constructs to Manchester Syntax which is supported by the OWL API.

OWL Subsumption and Disjointness Reasoning With the file properly created and loaded into the OWL API we can now use this file in combination with the hermit reasoner to infer a subsumption hierarchy of the different concept types. To get the subsumption hierarchy of a concept we use the provided getSubclasses method from the reasoning interface. A challenge in the inferring of hierarchies is that in contrast to the database we do not have a clear separation between concepts and other MDO constructs. For example in the owl file the constructs levels and concepts are seen as classes but we only need the concepts to be reasoned over. To accomplish this, we split the reasoning process into three parts reasoning the three main concept classes: entity concepts, dimensional concepts and multi-dimensional concepts. In the individual parts we iterate through all classes i.e. concepts and get their direct subclasses as shown in listing 23.

```
//select the class we want to infer over
OWLClass entityconcepts = df.getOWLClass(IRI.create("http://dke.uni-linz.
    ac.at/mdo#", "Entity"));
//get subclasses of the class (in this case we get a list of
    entityclasses)
Set<OWLClass> concepts = hermit.getSubClasses(entityconcepts, false).
    getFlattened();
//iterate through classes
for (OWLClass cl : concepts) {
  //check if valid class
  if (hermit.isSatisfiable(cl)) {
    //gets the subclasses of the entityclasses so in this case the
        concepts
    for (OWLClass sub : hermit.getSubClasses(cl, true).getFlattened()) {
        // if the entityclass has no concepts defined we do not need to
            reason over it
        if (!sub.isOWLNothing()) {
          //write hierarchy into database
          writeSubsumptionToDb(sub, cl, update, classprefix, conceptType)
              ; }   }        }
```

Listing 23: Inferring the subsumption hierarchy of concepts

In the course of implementing the prototype we also wanted to get the disjoint concepts for every concept. When conducting performance studies in section 4.6.3 we discovered that this approach, which gets all disjoint classes for every concept is, not exercisable as the processing time increases in an exponential manner. The implementation of disjointness inferring is very similar to subsumption inferring and shown in listing 24.

```
//select the class we want to infer over
OWLClass entityconcepts = df.getOWLClass(IRI.create("http://dke.uni-linz.
    ac.at/mdo#", "Entity"));
//get subclasses of the class (in this case we get a list of
    entityclasses)
Set<OWLClass> concepts = hermit.getSubClasses(entityconcepts, false).
    getFlattened();
//iterate through classes
for (OWLClass cl : concepts) {
disjointClasses = removeNonConceptClasses(hermit.getDisjointClasses(cl).
    getFlattened(), concepts);
        if (disjointClasses.size() > 0) { // if disjoint concepts are
            found write to DB
                for (OWLClass sub : disjointClasses) {
                        writeDisjointToDb(sub, cl, update, classprefix,
                            conceptType); } } }
```

Listing 24: Inferring the disjointness of concepts

Persisting the subsumption hierarchy in the MDO DB The writing of the data back into the database uses again JDBC. At this point it can happen that the reasoner has a hierarchy between owl-classes that are not entity concepts, for example the subsumption

hierarchy between an entity class and its entity concepts. Such non-concept relations are no problem as the constraints on the hierarchy tables, inside the MDO-DB, do not allow anything else than concepts being written into them.

In the ontology, the classes have the form 'http://dke.uni-linz.ac.at/mdo#conceptname'. To be able to write the subsumption hierarchy into the database we need to get ride of the prefix. The writing process is shown in listing 25.

```
// get the name of concepts without the classprefix
String sup = superClass.toString().replace(classprefix, "").replaceAll(">
    ", "");
String subs = subClass.toString().replace(classprefix, "").replaceAll(">"
    , "");

try {
 // write into the right subsumption table
 update.executeUpdate("INSERT␣INTO␣"+ mdoConstructPrefix +"
     _directsubsumption␣VALUES␣('" + subs + "',␣'" + sup + "')");
 }
 catch (SQLException ex) {
 //construct hierarchies of non-concept constructs are simple ignored
 if (ex.getSQLState().startsWith("23")) {
 }
 else {
  ex.printStackTrace();
 }
}
```

Listing 25: Writing the reasoning results into the database

4.6.3. Performance

As we already indirectly measured the performance of the mapping process with the ANTLR implementation in section 3.6, we only measure the Java part of the Reasoner implementation in this section i.e. the creating of the OWL file, reasoning over it, and the persisting of the subsumption hierarchy into the database.

While making our tests we discovered that deriving disjointness of classes, which we wanted to do in the same manner as creating the subsumption hierarchy, was not suitable after a certain amount of concepts. Figure 21 shows how the evaluation time of the reasoner rises exponentially. After inserting only 125 entity concepts the reasoning process already took over 5 minutes to complete. For this reason we omitted reasoning over more complex concept types. Figure 21 depicts the execution time for reasoning including disjointness and reasoning excluding disjointness.

Without reasoning over the disjointness of classes we get the results, shown in figure 22. As the results show we do not have a linear increase over insertion of more concepts, like the ANTLR part, but a squared one, which is still a big increase but not as unfavorable as the exponential increase. The reason for this behavior is that, in contrast to the ANTLR inserts or the DWH implementation, the reasoner cannot make a delta update by reasoning only over new created concepts. It has to create a new reasoning over all

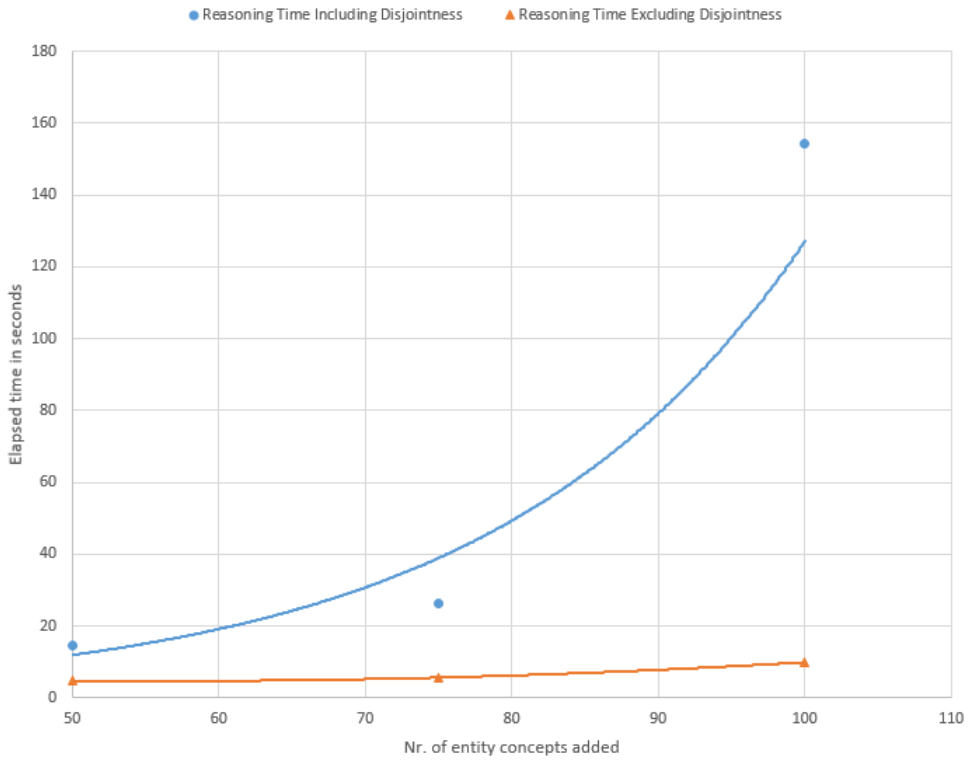

Figure 21: Comparison of the reasoning time including and excluding disjointness reasoning

concepts and additionally all parts of the MDO-Base e.g. Dimensions. We assumed the difference between the execution time of the different concept types would be even bigger because the reasoner always has to reason over the whole ontology but the difference seems to be rather small but getting bigger over the number of inserted concepts.

We also wanted to see if our implementation of the reasoning process has any major performance issues in comparison with professional software, as benchmark we compared the execution time of creating a subsumption hierarchy in the MDO reasoner with the execution time it takes Protégé to infer a subsumption hierarchy over the same ontology. The result of this benchmark is shown in figure 23. In this figure we see the MDO reasoner is slightly slower than Protégé. This might by due to the fact that the MDO reasoner has to create its OWL file first and has to conduct read and write operations on the MDO DB.

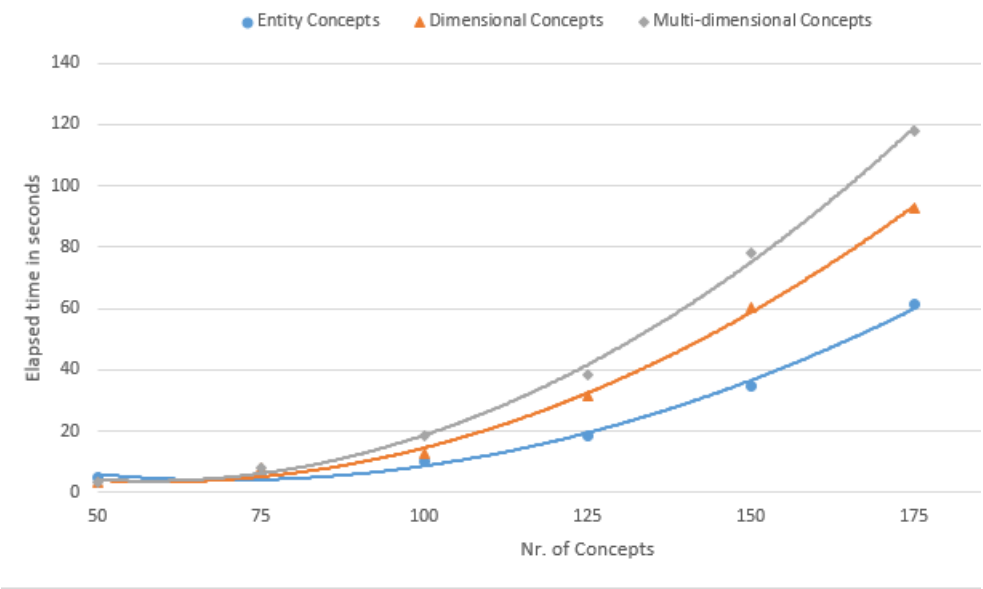

Figure 22: Runtime MDO Reasoner

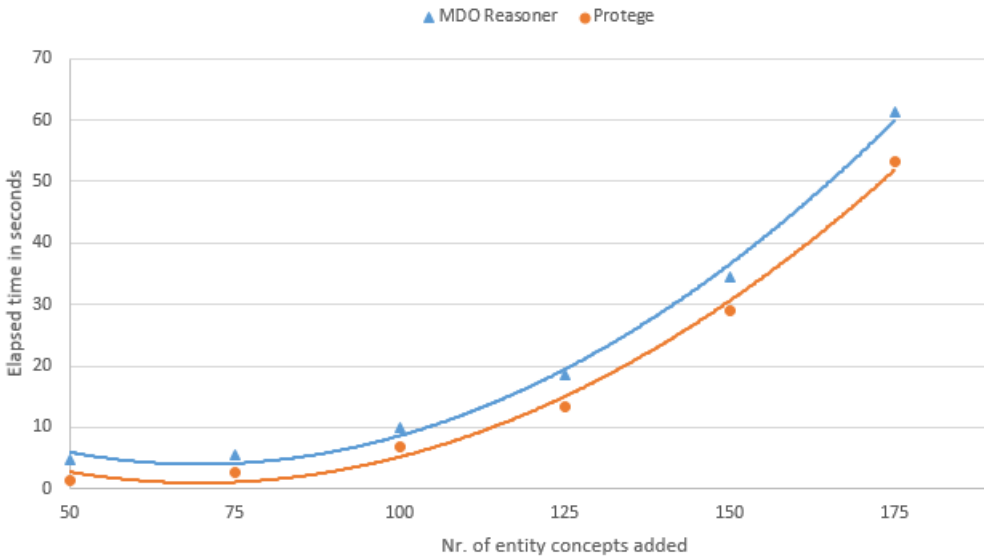

Figure 23: Comparison MDO Reasoner and Protégé

4.7. Discussion

As we have seen some problems occurred while creating our reasoner implementation which need to be addressed in future research. First our separation between single- and multi table concepts, in the other part of this thesis we solved the problem of multi table concepts not with a compound trigger but with a rather complicated query (shown in section 6.5.3). Between these two methods one is easier to understand (compound trigger) but the other may be be faster (query).

Another problem we found was the asserting of disjoint classes and that this should not be done over the whole ontology but rather on demand for specific concepts. In further work we could try to improve the reasoning process in the way that reasoning for single concepts could be possible.

Further we should check if we really want to create the mapping in the future with Manchester Syntax, although it is easier to read the user will hardly ever want to (or be able) change the mapping by hand.

5. MDO-DWH Mapping

In this section we will explain our semantic data warehouse(semDWH) and the mapping between the MDO and the semDWH. The semantic data warehouse and the data warehouse mapper were originally implemented by, Arjol Qeleshi, a member of the SemCockpit Project. In the course of this master thesis some details of the semDWH needed to be changed.

As shown in the global architecture (figure 4) we differentiate between two data warehouses. The enterprise data warehouse (EDWH) and the semDWH which contains a subset of the EDWH data, structured according to the generic semDWH structure (explained in this chapter) and enriched with concepts represented as views.

First we show the data structure of the semDWH and second how the MDO-DWH mapper maps the concepts from the MDO Syntax into concept views of the semDWH. At last we will look at the performance of our MDO-DWH mapper.

5.1. Overview

This section describes the basic data structure of the semantic data warehouse. An excerpt of the underlying data model of the data warehouse is shown in figure 2 at the beginning of the thesis.

Entities Every level in the model is represented by an entity table in the database, having the prefix e_. This entity tables represent the entity classes of the previous sections. The schema of a sample entity table is shown in listing 26.

```
CREATE TABLE e_province (
  province VARCHAR2(40)  PRIMARY KEY,
  inhabitants NUMBER(10) NOT NULL,
  sqkm NUMBER(8, 2) NOT NULL,
  inhpersqkm NUMBER(16, 10) NOT NULL,
);
```

Listing 26: Entity table in the semDWH

Entity classes that are used in different dimensions share their entity tables e.g. province in the insurant or doctor dimension.

Dimensions For every dimension in the data warehouse we added specific dimension tables, in total five. The first table is the dimension table with the naming scheme d_dimensionname. This table contains all nodes of the whole dimension.

Next are two level tables the d_dimensionname_ldr and d_dimensionname_lrr tables. LDR stands for level direct rollup, this table contains the name of the node and the level it directly rolls up to. LRR stands for level reflexive rollup and contains all levels a node rolls up to.

The next tables are called d_dimensionname_ndr and d_dimensionname_nrr. NDR is the node direct rollup table and contains nodes and their direct supernode. NRR contains nodes and all their super nodes.

Another rollup table is created out of the other tables called r_dimensionname. This table contains all valid rollup combinations and is used to make other queries easier. Table 24 shows some example values of the tables.

D_INSURANT

Insurant	Insurant_level
Insurant1	insurant
Insurant2	insurant
Amstetten	district
Burgenland	province

D_INSURANT_LDR

Insurant	Insurant_sl
Insurant1	district
Insurant2	district
Amstetten	province
Burgenland	all

D_INSURANT_NRR

Insurant	Insurant_sn
Insurant1	Insurant1
Insurant1	Amstetten
Insurant1	Niederösterreich

D_INSURANT_NDR

Insurant	Insurant_sn
Insurant1	Amstetten
Amstetten	Niederösterreich
Niederösterreich	all

Table 24: Dimension table examples

Dimension roles Dimension roles are represented inside the data warehouse by views (prefix dr_) that are a one to one mapping from the underlying base dimension to the dimension role. The only difference is that the columns are renamed to match the name of the dimension role. This is necessary because we work with natural joins where the column names need to match each other to work.

For example the creation of the dimension role acting doctor is shown in listing 27. In the listing we take the dimension d_doctor and make a view which renames the column 'doctor' to 'actingDoctor'. All dimension role tables are created in the same manner.

```
CREATE VIEW dr_actingDoctor AS
        SELECT doctor AS actingDoctor, doctor_lvl AS actingDoctor_lvl
        FROM d_doctor;
```

Listing 27: Example View for dimension role acting doctor

Dimensionspaces Every dimensionspace (e.g. ds_ambtreatmentspace in listing 28) is created as cartesian product of the dimensionroles of a dimensionspace. For every dimensionspace an additional rollupspace(e.g. rs_ambtreatmentspace in listing 28) is created. The rollupspace will later be used in the query definitions to make them easier.

```
CREATE or REPLACE VIEW ds_ambtreatmentspace
SELECT
  "ACTDOC",  "INSURANT",  "LEADDOC",  "MEDSERVITEM",  "TTIME"
FROM "DR_ACTDOC", "DR_INSURANT", "DR_LEADDOC", "DR_MEDSERVITEM", "
    DR_TTIME";

CREATE or REPLACE VIEW rs_ambtreatmentspace
SELECT
  "ACTDOC", "ACTDOC_LVL", "ACTDOC_SN", "ACTDOC_SL",
  "INSURANT", "INSURANT_LVL", "INSURANT_SN", "INSURANT_SL",
  "LEADDOC", "LEADDOC_LVL", "LEADDOC_SN", "LEADDOC_SL",
  "MEDSERVITEM", "MEDSERVITEM_LVL", "MEDSERVITEM_SN", "MEDSERVITEM_SL",
  "TTIME", "TTIME_LVL", "TTIME_SN", "TTIME_SL"
FROM "R_ACTDOC", "R_INSURANT", "R_LEADDOC", "R_MEDSERVITEM", "R_TTIME"
```

Listing 28: Example views for Dimensions/Rollupspaces

5.2. Entity Concepts

Concepts in general are represented in our semantic data warehouse as views. The specific representation of entity concept views is shown in this section. As already introduced in section 2.3.1 there are seven different supported entity concepts: primitive, sql-defined, nominal, attribute restricted, conjunctive, disjunctive and complement. Table 25 shows how these concepts can be represented as views, this representations are taken from [Neumayr et al., 2013, p.10].

Remember, the user has to be careful when creating an sql-defined concept. The query the user writes is not checked regarding feasibility this means the user can even add invalid queries which can cause problems in the data warehouse.

MDO Syntax	**Generated SQL(SemDWH)**
CREATE ENTITY CONCEPT *ec* FOR *ecl* AS PRIMITIVE; SQL *sqlquery*;	create materialized view *ec* AS *sqlquery*;
$\{entity_1, \ldots, entity_n\}$;	select *ecl* from *ecl* where *ecl* in ($entity_1$ ... $entity_n$);
attr θ *value*;	select *ecl* from *ecl* where *attr* θ *value*;
(ec_1 OR ... OR ec_n);	(select *ecl* from ec_1) union ... union (select *ecl* from ec_n);
(ec_1 AND ... AND ec_n);	(select *ecl* from ec_1) intersect ... intersect (select *ecl* from ec_n);
(NOT ec');	(select *ecl* from *ecl*) minus (select *ecl* from ec');

Table 25: Entity concept mapping to SQL

Table 26 shows how our running example is represented in data warehousing views. For the user to look at the mapping he or she has to access the sql_mapping table and filter by 'entityconcept'.

MDO Syntax	**SQL Representation**
	CREATE MATERIALIZED VIEW
create entity concept myProvince for province as province.Niederösterreich	"EC_MYPROVINCE" AS SELECT "PROVINCE" FROM "E_PROVINCE" WHERE "PROVINCE" IN ('Niederösterreich')
create entity concept YoungDoctor for doctor as (doctor.age >'35')	"EC_YOUNGDOCTOR" AS SELECT "DOCTOR" FROM "E_DOCTOR" WHERE "AGE" <'35'
create entity concept HighDensDistr for district as (district.HighPopulationDistrict and district.SmallDistrict)	"EC_HIGHDENSDISTR" AS (SELECT "DISTRICT" FROM "EC_HIGHPOPULATIONDISTRICT") INTERSECT (SELECT "DISTRICT" FROM "EC_SMALLDISTRICT")
create entity concept YoungOrOldDoc for doctor as (doctor.OldDoctor or doctor.YoungDoctor)	"EC_YOUNGOROLDDOC" AS (SELECT "DOCTOR" FROM "EC_OLDDOCTOR") UNION (SELECT "DOCTOR" FROM "EC_YOUNGDOCTOR")
create entity concept notSmallDistrict for district as (NOT district.SmallDistrict)	"EC_NOTSMALLDISTRICT" AS (SELECT "DISTRICT" FROM "E_DISTRICT") MINUS (SELECT "DISTRICT" FROM "EC_SMALLDISTRICT")

Table 26: Running example entity concepts in SQL

As we can see in the implementation we used materialized views over normal views for performance reasons. Also all concept views have a prefix according to the MDO construct they represent, for example 'EC_' for entity concepts.

5.3. Dimensional Concepts

Table 27 shows how dimensional concepts can be represented as views, this representations are taken from [Neumayr et al., 2013, p.11].

MDO Syntax	**SQL Representation**
CREATE DIMENSIONAL CONCEPT dc FOR d AT $fromLevel$[.. $toLevel$] AS	create materialized view dc AS
PRIMITIVE;	
SQL $sqlquery$;	$sqlquery$;
->$level$: ec;	select d from d_level where d in (select * from ec);
EXPAND dc';	select d from d_nrr where d_sn in (select d from dc');
dc' $level$;	select d from dc' natural join d_level;
(dc_1 OR ... OR dc_n);	(select d from dc_1) union ... union (select d from dc_n);
(dc_1 AND ... AND dc_n);	(select d from dc_1) intersect ... intersect (select d from dc_n);
NOT dc';	(select d from d) minus (select d from $dc1$);

Table 27: Dimensional concept mapping to SQL

As we can see in table 28 the MDO-DWH mapper ignores the signature we declare when specifying a new concept.

The result of a dimensional concept view is a set of nodes. With the dimension separated into the different tables the implementation of the different concepts is straightforward. For example, for the hierarchical expansion of a dimensional concept we use the node reflexive rollup table. The query for constructing a hierarchically expanded dimensional concept gets all nodes from the reflexive rollup table that have as super node the node from the concept we want to expand. The result from this query is the hierarchically expanded dimensional concept.

In the queries we see a table that we did not cover yet, the d_level table. For every dimension level exists a d_level table, for example for the level district of the dimension insurant exists the table d_insurant_district, that contains all nodes of the specific level. With these level tables the restriction of a dimensional concept to a certain level is simply the natural join of the level table and the concept table of the concept we want to restrict.

Table 28 shows how the running example is represented in data warehouse views. For the user to look at the mapping he has to access the sql_mapping table and filter by 'dimensionalconcept'.

MDO Syntax	**SQL Representation**
	CREATE MATERIALIZED VIEW
create dimensional concept SqlMayer for Doctor at Doctor.doctor AS sql 'select doctor from d_doctor where doctor = 'Mayer;'	"DC_SQLMAYER" AS select doctor from d_doctor where doctor = 'Mayer';
create dimensional concept byECHighDensDis for Insurant AT Insurant.district AS ->(Insurant.district:district.HighDensDistr);	"DC_BYECHIGHDENSDIS" AS SELECT l."INSURANT" FROM "L_INS_DISTRICT" l, "EC_HIGHDENSDISTR" e WHERE l."INSURANT" = e."DISTRICT";
create dimensional concept InHighDensDis for Insurant AT Insurant.insurant .. Insurant.district AS expand Insurant.byECHighDensDis;	"DC_INHIGHDENSDIS" AS SELECT r."INSURANT" FROM "DC_BYECHIGHDENSDIS" dc, "D_INSURANT_NRR" r WHERE dc."INSURANT" = r."INSURANT_SN";
create dimensional concept InsInHighDensDis FOR Insurant AT Insurant.insurant AS Insurant.InHighDensDis Insurant.insurant;	"DC_INSINHIGHDENSDIS" AS SELECT dc."INSURANT" FROM "DC_INHIGHDENSDIS" dc NATURAL JOIN "L_INS_INSURANTS";
create dimensional concept notHighDensity for Insurant at Insurant.district as (NOT Insurant.byECHighDensDis);	"DC_NOTHIGHDENSITY" AS (SELECT "INSURANT" FROM "D_INSURANT") MINUS (SELECT "INSURANT" FROM "DC_BYECHIGHDENSDIS");
create dimensional concept YoungInsInHighDens for Insurant at Insurant.insurant as (Insurant.byECYoungInsurant and Insurant.InsInHighDensDis);	"DC_YOUNGINSINHIGHDENS" AS (SELECT "INSURANT" FROM "DC_INSINHIGHDENSDIS") INTERSECT (SELECT "INSURANT" FROM "DC_BYECYOUNGINSURANT");
create dimensional concept YoungInsOrHighDens for Insurant at Insurant.insurant as (Insurant.byECYoungInsurant or Insurant.InsInHighDensDis);	"DC_YOUNGINSINHIGHDENS" AS (SELECT "INSURANT" FROM "DC_INSINHIGHDENSDIS") UNION (SELECT "INSURANT" FROM "DC_BYECYOUNGINSURANT");

Table 28: Running example dimensional concepts in SQL

5.4. Multi-dimensional Concepts

Table 29 shows how multi-dimensional concepts can be represented as views, this representations are taken from [Neumayr et al., 2013, p.14].

MDO Syntax	**SQL Representation**
CREATE MULTIDIMENSIONAL CONCEPT *mc* FOR *ds* AS	create view mc AS
PRIMITIVE; SQL *sqlquery*;	*sqlquery*;
->(*dr*:*dc*);	select * from *ds* where *dr* in (select * from *dc*);
EXPAND mc';	select * from *rs* s, mc' c where (s.dr_1 = c.dr_1) and ... and (s.dr_n =c.dr_n);
$mc'[ds]$;	select * from mc' natural join *ds*;
(mc_1 OR ... OR mc_n);	(select * from mc_1) union ... union (select * from mc_m);
(mc_1 AND ... AND mc_n);	select * from mc_1 natural join ... natural join mc_m;
(NOT mc');	(select * from *ds*) minus (select * from mc');

Table 29: Multi-dimensional concept mapping to SQL

At the hierarchy expansion concept we see why we need the roll-up spaces from section 5.1, we simple join these views together and get the hierarchically expanded concept view. One interesting thing to note is the mapping of conjunctive multi-dimensional concepts. Contrary to entity concepts and dimensional concepts we do not create them by using an intersection but by using a natural join, this leads to an interesting behavior, if one is not aware of that fact. In our running example our conjunctive concept consists of one concept that is comprised of all high density districts of acting doctors and the other concept is comprised of all high density districts of insurants. If we would use an intersection here, the result of this conjunction would be empty, but since we used a natural join the resulting conjunctive multi-dimensional concept is the cartesian product of these two concepts.

Table 30 shows the concept views from the running example. For the user to look at the mapping she has to access the sql_mapping table and filter by 'multidimensionalconcept'.

MDO Syntax	**SQL Representation**
	CREATE VIEW
create multidimensional flat concept byDcHighDens FOR Ins_Distr AS ->(insurant:Insurant.byECHighDensDis)	"MC_BYDCHIGHDENS" AS SELECT ds.* FROM "DS_INS_DISTR" ds, "DC_BYECHIGHDENSDIS" dc WHERE ds."INSURANT" = dc."INSURANT";
create multidimensional hierarchic concept expandDocInsHPC FOR Ins_Act AS expand DocInsInHPC;	"MC_EXPANDDOCINSHPC" AS SELECT rs."ACTDOC", rs."INSURANT" FROM "MC_DOCINSINHPC2" ds, "RS_INS_ACT" rs WHERE ds."ACTDOC" = rs."ACTDOC_SN" AND ds."INSURANT" = rs."INSURANT_SN";
create multidimensional flat concept restrDocInsHPC FOR Ins_Act_persons AS expandDocInsHPC [Ins_Act_persons];	"MC_RESTRDOCINSHPC" AS SELECT * FROM "DS_INS_ACT_PERSONS" ds NATURAL JOIN "MC_EXPANDDOCINSHPC" mc;
create multidimensional flat concept notHighDens FOR Ins_Distr AS (not byDcHighDens);	"MC_NOTHIGHDENS" AS (SELECT * FROM "DS_INS_DISTR") MINUS (SELECT * FROM "MC_BYDCHIGHDENS");
create multidimensional flat concept DocInsInHPC FOR DocIns_Distr AS (byDcDocHPC and byDcInsHPC);	"MC_DOCINSINHPC" AS SELECT * FROM "MC_BYDCDOCHPC" NATURAL JOIN "MC_BYDCINSHPC";
create multidimensional flat concept YoungOrHighDensInsurants FOR Ins_Insurants AS (byDcYoungIns or restHighDensInsurants);	CREATE VIEW "MC_YOUNGORHIGHDENSINSURANTS" AS (SELECT * FROM "MC_BYDCYOUNGINS") UNION (SELECT * FROM "MC_RESTHIGHDENSINSURANTS");

Table 30: Running example multi-dimensional concepts in SQL

5.5. Prototype Implementation and Performance

Here we will show a brief overview about how the DWH-mapper operates. The DWH-Mapper itself exists within an PL\SQL packages which is structured into multiple smaller packages, one package for every major mdo-construct e.g. a package for entity concepts, another for dimensional concepts etc. When a concept is inserted or altered in the MDO-DB, a trigger calls the right function of the package. The function then generates the sql-mapping code for the specific concept. The in this manner generated SQL representations of MDO concepts are then created in the semDWH.

The MDO-DWH mapper implements three triggers on every kind of concept, a before statement trigger, an after each row trigger and an after statement trigger. Sample triggers are shown in listing 29.

```
TRIGGER "Before␣Statement␣Trigger"
      BEFORE DELETE OR INSERT OR UPDATE ON "dc_conjunctive"
      BEGIN
        DELETE FROM sql_pending_mapping;
      END;
TRIGGER "After␣Row␣Trigger"
      AFTER DELETE OR INSERT OR UPDATE ON "dc_conjunctive"
      FOR EACH ROW
      DECLARE
        mdo_id VARCHAR2(200);
      BEGIN
        IF deleting THEN
          mdo_id := :old."ID"; ELSE
          mdo_id := :new."ID";
        END IF;
        INSERT
          /*+ IGNORE_ROW_ON_DUPKEY_INDEX(
            sql_pending_mapping,sql_pending_mapping_pk
          ) */
          INTO sql_pending_mapping (mdo_construct, mdo_id)
          VALUES ('entityconcept', mdo_id);
      END;
TRIGGER "After␣Statement"
      AFTER DELETE OR INSERT OR UPDATE ON "dc_conjunctive"
      BEGIN
        INSERT INTO sql_mapping_sequence
          (sequence_nr, mdo_construct, mdo_id)
          SELECT
            sql_dwh_obj_seq_nr.nextval, mdo_construct, mdo_id
          FROM
            sql_pending_mapping;
        DELETE FROM sql_pending_mapping;
      END;
```

Listing 29: DWH mapping Trigger on conjunctive dimensional concepts

The purpose of these triggers is to maintain a valid state of the database. The DWH mapper logs all concepts that are being created over time. The user has the possibility

to go to a specific point in time and set the state of the semDWH to this point, for this purpose the mapper needs the sql_mapping_sequence tables. The sql_pending_mapping table keeps track of all instances affected, to be processed by the DWH mapper upon completion of the statement.

The real mapping of the MDO-constructs to the semDWH SQL code happens in the after statement trigger when the trigger inserts the data into sql_mapping_sequence. At insertion an after row trigger is triggered that calls the right mapping function for the MDO-concept. The trigger code is shown in listing 30.

```
TRIGGER dwm_br_automap
      BEFORE INSERT ON sql_mapping_sequence
      FOR EACH ROW
      DECLARE
        stmt_id INTEGER;
        ddlcode dwh_mapper_frontend.DDLPAIR;
      BEGIN
        IF (dwh_mapper_frontend.automap_ddl_for_mdo) THEN
          ddlcode := dwh_mapper_frontend.generate_ddl_mapping(
            :new.mdo_construct,
            :new.mdo_id
          );
          stmt_id := dwh_mapper_frontend.generate_ddl_mapping_stmts(
            ddlcode.create_ddl
          );
          :new.sql_create_code := stmt_id;
          stmt_id := dwh_mapper_frontend.generate_ddl_mapping_stmts(
            ddlcode.drop_ddl
          );
          :new.sql_drop_code := stmt_id;
        END IF;
      END;
```

Listing 30: Automap Trigger to create the SQL mapping

The generate_ddl_mapping functions has as parameter the ID of the construct and the type (entity concept,dimensional concept, multi-dimensional concept). With this parameters it performs a lookup in the mapping_strategy table which contains the function that needs to be executed to make the mapping. This SQL-mapping code can be reviewed by the user in a special view called 'sql-mapping'. The real execution of the SQL code in the semantic data warehouse (i.e.the creating of views) has to be invoked by the user. For this purpose the MDO-DWH mapper contains a frontend package function (called generate dwh objects) which starts the creation of the data warehouse concepts. Because of the restrictions of the oracle database (only 30 characters per table name) the DWH-Mapper also contains a function that automatically asserts a new name to a concept in the data warehouse, if the proposed name is not suitable.

Performance Here we measure how long it takes the semantic data warehouse to execute the generated SQL code, i.e. to create the views in the semDWH.

The creation of concept views in the semDWH has like the ANTLR inserts a linear increase. An interesting behavior we see when we look at figure 24 is the time it takes the data warehouse to create multi-dimensional concepts, which is significantly lower than the other two concept types. This is because mutli-dimensional concepts are not materialized. We see a satisfactory performance for the creation of the concept views inside the semDWH.

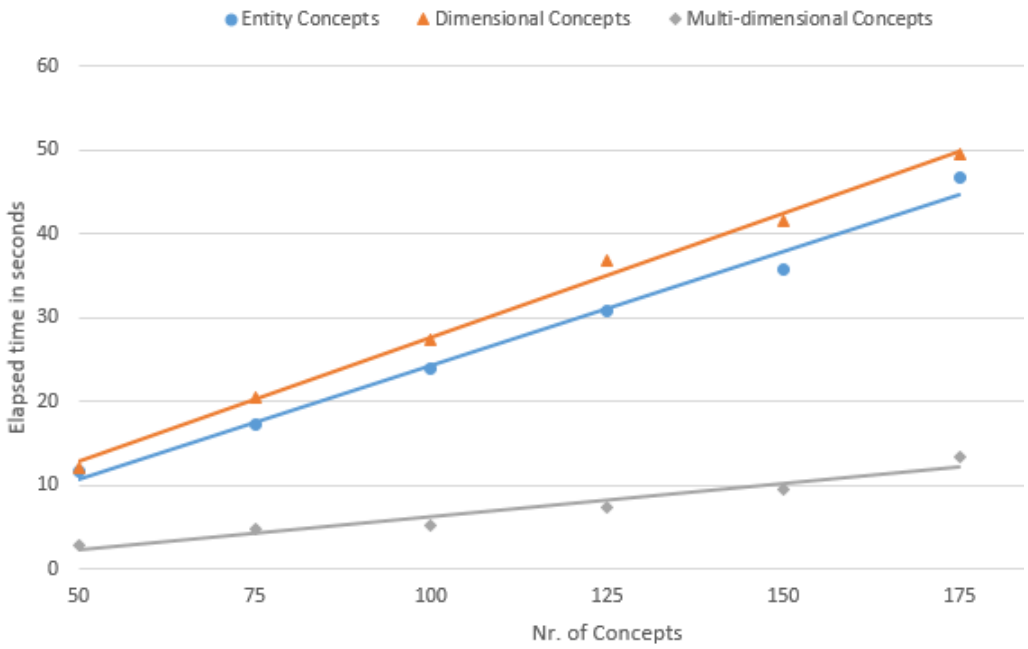

Figure 24: DWH measurement results

5.6. Discussion

The semantic data warehouse works as desired and with an acceptable speed. The only part that is still problematic is the initial transformation from a normal data warehouse into the semantic data warehouse. This transformation has still to happen by hand, meaning we have to manually write procedures to transform the data warehouse. In further projects we should look into automating this process to make our approach easier to use. Other than that our mapping process seems to work appropriately fast.

6. Contextualized Concepts

In this section we will describe the implemented of contextualized concept. The section is structured as follows. First we look at the data structure of the new concept types in the MDO-DB. Then we show how we implemented the concrete syntax for contextualized concepts with ANTLR, afterwards we look how the OWL mapping works, and last we will see how the contextualized concepts are being mapped into the semantic data warehouse. To create the contexts necessary for our concepts we used node-concept-level dimensional concepts and point- multi-dimensional concepts. These concepts where not covered in [Neumayr et al., 2013] and therefore we will also show their OWL and SQL mapping in sections 6.4 and 6.5. Contextualized concepts are also an issue for [Steiner et al., 2015] who use contextualized concepts in the context of contextualized rule evaluation.

Because the implementation of dimensional and multi-dimensional concepts is very similar we will mainly describe the implementation of dimensional concepts and show the difference to multi-dimensional concepts if necessary.

6.1. Overview

Contextualized concepts are a special type of concepts and are briefly mentioned in [Neuböck et al., 2014, p.16]. They consist of two parts, contextualized concepts and contextspecific concepts. Following [Neuböck et al., 2014, p.16] we implemented the concepts using the abstract superclass rule, meaning the contextualized concept acts as the abstract super class that defines the signature for the method, the contextspecific concepts than act as concrete implementation. So a contextualized concepts consists of multiple context specific concepts, these context specific concepts consist of a concept that is only valid for a specific context e.g. only for certain nodes of a dimension or points of a dimensionspace. An example depiction of the structure of a contextualized concept is show in figure 25. We can see the contextualized concept oldDoctor which consists of five contextspecific concepts. These concepts define, for their respective contexts, when a doctor is considered being an old doctor. This example will also be our example to show how we create contextualized concepts.

For this concept to work we have the prerequisite that the contexts of one contextualized concept on one hierarchy level are disjoint to one another so that on a distinct level a point/node belongs to exactly one most specific context.

6.2. Extending the MDO DB with Contextualized Concepts

Contextualized and contextspecific concepts are new kinds of concepts extending the already existing kinds of concepts. The MDO-DB structure for dimensional contextualized concepts is shown in figure 26. Figure 26 contains the UML class diagram and a small object diagram for dimensional contextualized concepts. The corresponding structure for multi-dimensional concepts would look very similar. A contextualized concept consists of multiple context specific concepts. Context specific concepts refer to a concept

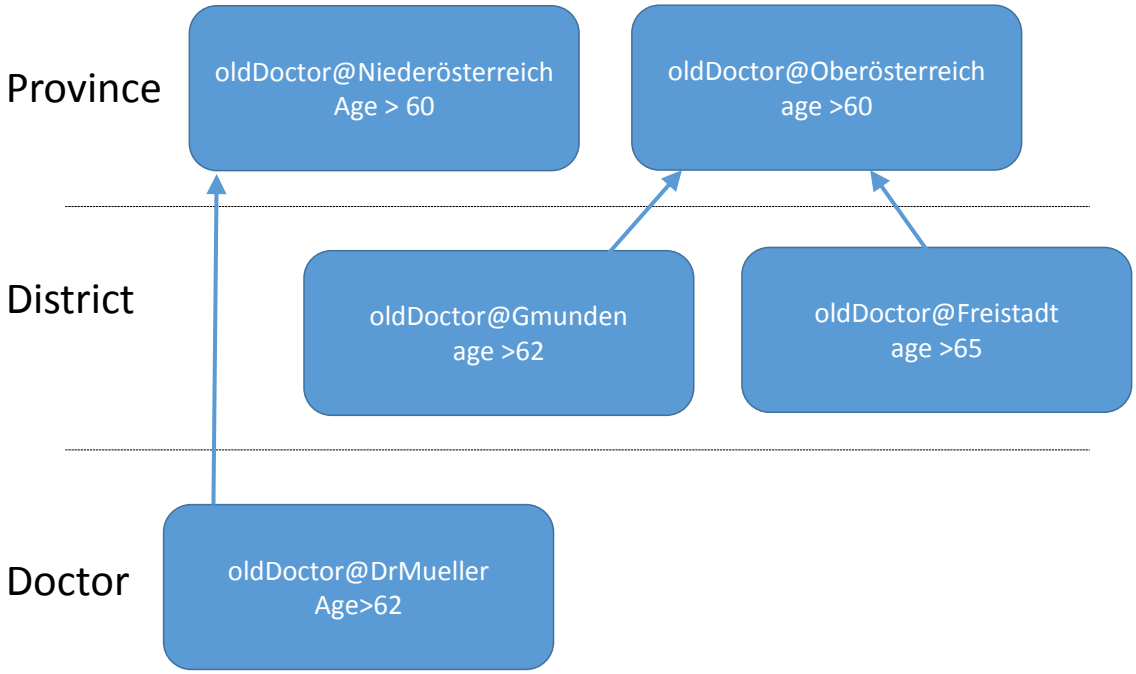

Figure 25: Context-specific concepts of contextualized concept oldDoctor

and have a context for which this concept is valid. A subsumption hierarchy over contexts is inferred by the reasoner and helps in the later stage to create the contextualized concepts. The connection between contextualized concepts and their context specific concepts is made by a third table called 'contextspecific for contextualized'. Listing 31 shows the DDL and listing 32 the DML statements for creating contextualized concepts. In the DML statements we only show one insert per table type to keep the listing short. To incorporate the new concepts into the existing MDO-DB the base class definition discriminator has to be extended with the newly created concepts. The concept tables for contextualized concepts follow the same implementation design as the old concept tables described in section 2.3.

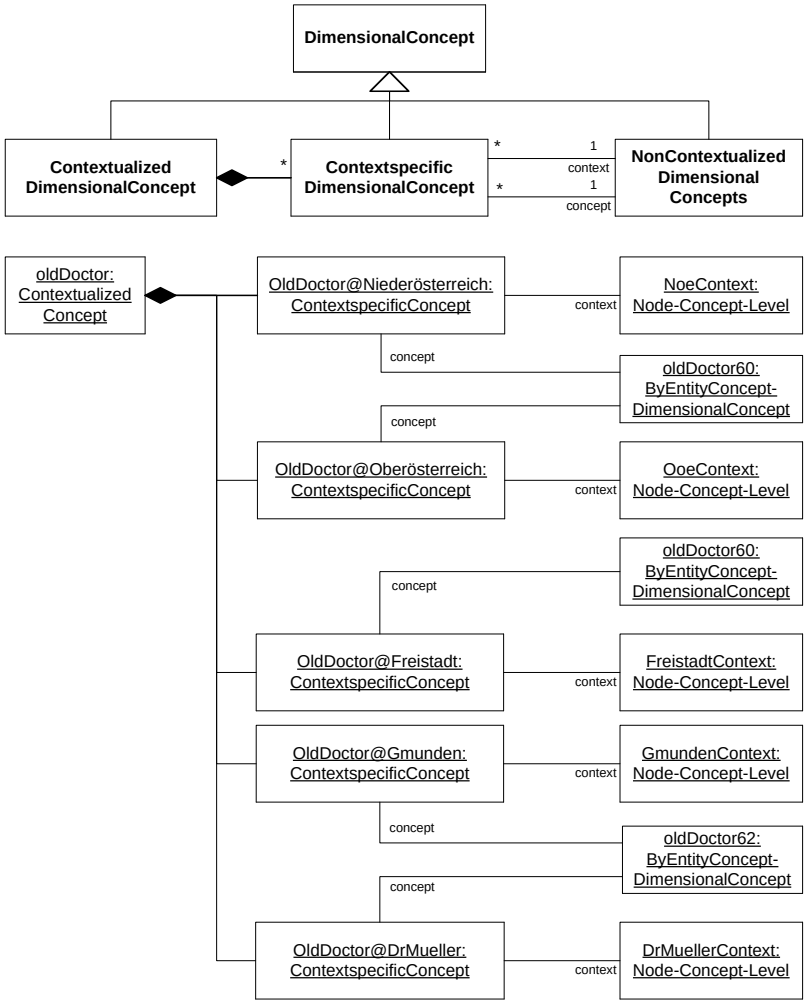

Figure 26: Overview over contextualized concepts class diagram (top) and object diagram(bottom)

```
1  CREATE TABLE dc_contextualized(
2      dcID varchar2(201) PRIMARY KEY ,
3      CONSTRAINT dc_contextualized_dc_fk FOREIGN
           KEY(dcID)
4      REFERENCES dc_dimconcept(dcID)
5  );
6  CREATE TABLE dc_contextspecific(
7      dcID varchar2(201) PRIMARY KEY,
8      contextspecific_dcID varchar2(201) not null,
           --global concept e.g. oldDoctor65
9      context_dcID varchar2(201) not null, -- the
           context it will be applied to e.g.
           lowerAustria
10     CONSTRAINT dc_local_dc_fk FOREIGN KEY(dcID)
11     REFERENCES dc_dimconcept(dcID),
12     CONSTRAINT dc_local_dcid_fk FOREIGN KEY(
           context_dcID)
13     REFERENCES dc_dimconcept(dcID),
14     CONSTRAINT dc_ctxtsp_dcid_fk FOREIGN KEY(
           contextspecific_dcID)
15     REFERENCES dc_dimconcept(dcID)
16 );
17 CREATE TABLE dc_contextspecforcontextlzdT(
18     dcID varchar2(201) not null, --dcID for the
           contextualized Concept
19     contextspecific_dcID varchar2(201) not null,
           --contextspecific concept
20
21     CONSTRAINT dc_localforcon_pkT PRIMARY KEY(
           dcID,contextspecific_dcID),
22     CONSTRAINT dc_localforcon_dc_fkT FOREIGN KEY
           (dcID)
23     REFERENCES dc_contextualizedTEMP(dcID),
24     CONSTRAINT dc_localforcon_conzdcid_fkT
           FOREIGN KEY(contextspecific_dcID)
25     REFERENCES dc_contextspecificTEMP(dcID)
26 );
```

Listing 31: DDL statements for creating contextualized dimensional concepts

```
1  INSERT INTO dc_dimconcept (dimensionId,
       dimconceptName, signature_discriminator,
       definition_discriminator)
2    VALUES ('Doctor', 'NoeContext', 'flatsignature
       ', 'nodeconceptlevel')
3  INSERT INTO dc_flatSignature (dcID, levelID)
4    VALUES ('Doctor.NoeContext', 'Doctor.doctor')
5  INSERT INTO dc_nodeConceptLevel (dcID, nodeID,
       slice_dcID, levelID)
6    VALUES ('Doctor.NoeContext', 'Doctor.province.
       Niederoesterreich', 'Doctor.
       DocDuplication', 'Doctor.doctor')
7  ----contextualized
8  INSERT INTO dc_dimconcept (dimensionId,
       dimconceptName, signature_discriminator,
       definition_discriminator)
9    VALUES ('Doctor', 'oldDoctorCtxlzd', '
       flatsignature', 'contextualized')
10 INSERT INTO dc_flatSignature (dcID, levelID)
11   VALUES ('Doctor.oldDoctorCtxlzd', 'Doctor.
       doctor')
12 INSERT INTO dc_contextualized (dcID)
13   VALUES ('Doctor.oldDoctorCtxlzd')
14 ----contextspecific
15 INSERT INTO dc_dimconcept (dimensionId,
       dimconceptName, signature_discriminator,
       definition_discriminator)
16   VALUES ('Doctor', 'oldDoctorCtxlzd1', '
       flatsignature', 'contextspecific')
17 INSERT INTO dc_flatSignature (dcID, levelID)
18   VALUES ('Doctor.oldDoctorCtxlzd1', 'Doctor.
       doctor')
19 INSERT INTO dc_contextspecific (dcID,
       contextspecific_dcID, context_dcID)
20   VALUES ('Doctor.oldDoctorCtxlzd1', 'Doctor.
       OldDoctor62', 'Doctor.doctorContext')
21 ----contextspecific for contextualized
22 INSERT INTO dc_contextspecforcontextlzd (dcID,
       contextspecific_dcID)
23   VALUES ('Doctor.oldDoctorCtxlzd', 'Doctor.
       oldDoctorCtxlzd1')
```

Listing 32: DML statements for creating contextualized dimensional concepts

6.3. Extending the MDO Parser with Contextualized Concepts

In this section we will show the grammar for extending the MDO parser with contextualized dimensional and multi-dimensional concepts. The grammar is shown in listing 33.

```
1  cdExpr
2      :  cdByEntityConceptName  | .......  | cdContextualized
3      ;
4
5  cdContextualized
6      : dimConceptName AT_SIGN context (COMMA dimConceptName AT_SIGN
           context)*;
7
8  cmExpr
9      :  cmByDimensionalConcept|  | .......  | cmContextualized
10     ;
11
12 cmContextualized
13     : mdConceptName AT_SIGN mdContext (COMMA mdConceptName AT_SIGN
           mdContext)*;
14
15 AT_SIGN: '@';
16 COMMA: ',';
```

Listing 33: Contextualized concept grammar for dimensional and multi-dimensional concepts

In order to make the creation of new contextualized concepts more easy to write we chose to instantiate our contextspecific concepts implicitly. Thats why we only see a parser rule definition for contextualized concepts (listing 33 lines 5 and 12) and not for contextspecific concepts . To extend the old parser rules with the new concepts alternatives for the new kinds of dimensional and multi-dimensional concepts needs to be added to the concept definition rules (lines 2 and 9)

The contextualized concept oldDoctor that will be created is, as already mentioned, depicted in figure 25. This contextualized concept consists of contextspecific definitions for Lower Austria(Niederösterreich), Upper Austria(Oberösterreich), Gmunden, Freis-

tadt and one specific doctor from Lower Austria. Table 31 shows the MDO Syntax and DML statements for creating the contextualized concept.

MDO Syntax	**Generated SQL(MDO-DB)**
CREATE DIMENSIONAL CONCEPT *contextualized* FOR *dimension* AT *level* AS $concept_1$@$context_1$ ... $concept_n$@$context_n$	INSERT INTO dc_contextualized (dcID) VALUES (*contextualized*) INSERT INTO dc_contextspecific (dcID, contextspecific_dcID, context_dcID) VALUES (*contextualized*1, $concept_1$,$context_1$) ... INSERT INTO dc_contextspecific (dcID, contextspecific_dcID, context_dcID) VALUES (*contextualizedn*, $concept_n$,$context_n$) INSERT INTO dc_contextspecforcontextlzd (dcID, contextspecific_dcID) VALUES (*contextualized*, *contextualized*1) ... INSERT INTO dc_contextspecforcontextlzd (dcID, contextspecific_dcID) VALUES (*contextualized*, *contextualizedn*)

Table 31: MDO Syntax and DML statements for creating a contextualized concept

As a naming convention our implicitly defined contextspecific concepts have the same name as the contextualized concepts with a running number appended on the end. Table 32 shows the DML statements for creating the contextualized concept.

MDO Syntax	**Generated SQL (MDO-DB)**
create dimensional concept oldDoctorCtxlzd FOR Doctor AT Doctor.doctor AS OldDoctor60@NoeContext, OldDoctor60@OoeContext, OldDoctor62@GmundenContext, OldDoctor65@FreistadtContext, OldDoctor62@doctorContext	INSERT INTO dc_contextualized (dcID) VALUES ('Doctor.oldDoctorCtxlzd') INSERT INTO dc_contextspecific (dcID, contextspecific_dcID, context_dcID) VALUES ('Doctor.oldDoctorCtxlzd1', 'Doctor.OldDoctor62', 'Doctor.doctorMuellerContext') INSERT INTO dc_contextspecforcontextlzd (dcID, contextspecific_dcID) VALUES ('Doctor.oldDoctorCtxlzd', 'Doctor.oldDoctorCtxlzd1') ... INSERT INTO dc_contextspecific (dcID, contextspecific_dcID, context_dcID) VALUES ('Doctor.oldDoctorCtxlzd5', 'Doctor.OldDoctor60', 'Doctor.NoeContext') INSERT INTO dc_contextspecforcontextlzd (dcID, contextspecific_dcID) VALUES ('Doctor.oldDoctorCtxlzd', 'Doctor.oldDoctorCtxlzd5')

Table 32: MDO Syntax and DML statements for example contextualized concept

The grammar for creating dimensional or multi-dimensional concepts stayed the same (grammar is shown in appendix B). To adjust our existing grammar we just needed to add a new option at cdExpr/mcExpr, add the new rule for contextualized concepts, and ANTLR automatically generates our new listener method. We omit showing the new methods for contextualized concepts and their SQL generation as they do not show any

new insights. The interested reader can go back to section 3.6 to see how we implement listener methods.

6.4. Extending the MDO Reasoner with Contextualized Concepts

In this section we will look at the OWL mapping for contexts, contextspecific concepts and contextualized concepts.

6.4.1. Contexts

Node-Concept-Level Concepts Node-Concept-Level concepts (NCL) where not part of [Neumayr et al., 2013] but a small part of [Neuböck et al., 2014]. In both papers no mapping to OWL is mentioned. With this concept we have the problem that the reasoner, that is located outside the data warehouse, does not know about the nodes inside the data warehouse and can therefore not infer any subsumption hierarchies over them. We solved this problem by creating the nodes we needed per hand and assigning the roll-up hierarchy. With this assigned roll-up hierarchy the reasoner was able to infer the subsumption hierarchy between the NCL concepts we created. We then used the NCL concepts as our representation for contexts in dimensional concepts. The OWL mapping for NCL concepts is shown in table 33 and example is shown in table 34. Figure 27 shows how our reasoner inferred the subsumption hierarchy of our contexts.

MDO Syntax	**OWL (Manchester Syntax)**
create dimensional concept *context* FOR *dimension* AT *level* AS *node*;	Class:*context* EquivalentTo: *dimension* and atLevel value *level* and rollsUpTo value *node*
create multidimensional flat concept FOR *ds* AS *point*	Class:*PCG* SubclassOf: *ds* EquivalentTo: *ds* and ((dr_1 rollsUpTo value $node_1$) AND ... AND (dr_n rollsUpTo value $node_n$))

Table 33: OWL mapping for node-concept-level and point-concept-granularity context

MDO Syntax	**OWL (Manchester Syntax)**
create dimensional concept NoeContext FOR Doctor AT Doctor.doctor AS province.Niederösterreich;	Class: Doctor.NoeContext EquivalentTo: Doctor and rollsUpTo value Doctor.province.Niederösterreich and atLevel value Doctor.doctor

Table 34: Example OWL mapping for node-concept-level context

Point-Concept-Granularity Point-Concept-Granularity (PCG) concepts where like NCL concepts not covered by [Neumayr et al., 2013] and only briefly mentioned in [Neuböck et al., 2014]. They are the multidimensional context counterpart to NCL concepts in dimensional contextualized concepts. Like node-concept-level concepts we have the problem that we do not know from the data warehouse how the different nodes of a point

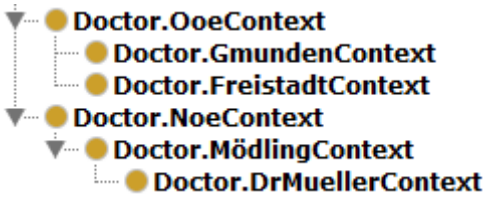

Figure 27: Context Hierarchy

roll-up to each other. Again we assigned the roll-up hierarchy of the nodes of a point by ourself and the reasoner could then use it in the further course of the project. The mapping for PCG Concepts is shown in table 33. The OWL mapper automatically splits the specified Point into its dimension roles and nodes.

Other Context Concepts In general every dimensional or multi-dimensional concept can be used as a context for contextualized concepts. The reason we chose NCL and PCG concepts was that they are very convenient because we only need one concept to create a context. Table 35 shows the difference between using a node-concept-level concept and another form for creating the concept containing doctors in Lower Austria. In using NCL and PCG as contexts we also followed the recommendation from [Neuböck et al., 2014, p.17].

DML statements (MDO-DB)
INSERT INTO dc_nodeConceptLevel (dcID, nodeID, slice_dcID, levelID) VALUES ('Doctor.NoeContext', 'Doctor.province.Niederösterreich', 'Doctor.DocDuplication', 'Doctor.doctor')
INSERT INTO EC_NOMINAL values('province.niederösterreich'); INSERT INTO EC_NOMINAL_ENTITY values ('province.niederösterreich', 'province.Niederösterreich'); INSERT INTO dc_byentityconcept(dcid, ecid, levelid) VALUES ('Doctor.NoeDim', 'province.niederösterreich', 'Insurant.province'); INSERT INTO dc_hierarchyexpansion(dcid, tobeexpanded_dcID) VALUES ('Doctor.NoeDimAll', 'Insurant.NoeDim'); INSERT INTO DC_CONCEPTLEVEL(dcid, slice_dcID, LEVELID) VALUES ('Doctor.NoeDoctors','Insurant.NoeDimAll', 'Doctor.doctor');

Table 35: Context inserts comparison between Node-Concept-Level Concept (Top) and the same concept built with other concepts (Bottom)

6.4.2. Contextspecific Concepts

Contextspecific concepts consist of a concept and a context on which this concept is applied. The mapping of contextspecific concepts is very straightforward and shown in table 36. We only need to make a conjunction of the the context and the concept that

is referred by the context. Like other dimensional concepts, contextspecific concepts are defined for a certain dimension at a certain levelrange, these are inherited from the contextualized concept.

An example contextspecific concept is shown in table 37, keep in mind that contect-specific concepts are defined implicitly. The name of a contextspecific concept is the name of their contextualized concept with a running number appended at the end.

MDO Syntax	**OWL (Manchester Syntax)**
concept@context	Class:$contextualized_n$ EquivalentTo: *dimension* and atLevel *levelrange* and *concept* and *context*

Table 36: Contextspecific OWL mapping

MDO Syntax	**OWL (Manchester Syntax)**
OldDoctor60@Doctor.NoeContext	Class: Doctor.OldDoctor1 EquivalentTo: Doctor and atLevel some DoctorFrdoctorTodoctor and Doctor.OldDoctor60 and Doctor.NoeContext

Table 37: Example contextspecific OWL mapping

6.4.3. Contextualized Concepts

With contexts and contextspecific concepts mapped we can now look at contextualized concepts. Contextualized concepts consist of multiple contextspecific concepts. These contextspecific concepts are valid for a certain context. Between these contexts we have a subsumption hierarchy and the most specific context is the valid concept for our Node or Point. To get the most specific concept we have some challenges. First, we need a subsumption hierarchy between the contexts, which we already solved. Second, the reasoner only shows direct subsumption relations, this can be a problem when a concept is only indirectly subsumed by another concept, this is shown in our test concept as the relationship between DrMueller and LowerAustria. To solve this problem we used the hierachical query[18] functionality in Oracle. Third we need to check if the subsumed concepts are really used in the contextualized context, as contextspecific concepts can be used in multiple queries. To test this case we inserted the context Mödling which is not part of our contextualized concept. All these problems lead to the rather complicated looking query shown in listing 34. We dissected the query to make it easier to understand. The query consists of three main parts. Part one (line 2) is a subquery responsible for getting all contextspecific concepts associated with a contextualized concept. Part two (line 7) is a subquery that acquires the contexts for the different contextspecific concepts and builds the excluding 'and not' statements, part two uses the subquery SQ1. Part three (line 12) lists all contextspecific concepts of a contextualized concepts and connects them with an 'OR', part three uses SQ2.

[18]Further information see `http://docs.oracle.com/cd/B19306_01/server.102/b14200/queries003.htm`

```
=SQ1
select DC_CONTEXTSPECIFICTEMP.CONTEXT_DCID
from dc_contextspecforcontextlzdT
join DC_CONTEXTSPECIFICTEMP on DC_CONTEXTSPECIFICTEMP.dcid =
    dc_contextspecforcontextlzdT.CONTEXTSPECIFIC_DCID
where dc_contextspecforcontextlzdT.DCID = ct.dcID
=SQ2
select listagg (' AND NOT ' || sub_dcid,'')
within group (order by sub_dcid) from DC_DIRECTSUBSUMPTION where
sub_dcID IN (SQ1) start with sup_dcID = cs.CONTEXT_DCID
CONNECT BY PRIOR sub_dcid = sup_dcid
=SQ3
select LISTAGG ( '(' || ct.CONTEXTSPECIFIC_DCID || SQ2 , ' OR ' )
within group (order by cs.dcID)
from DC_CONTEXTSPECIFICTEMP cs, dc_contextspecforcontextlzdT ct
WHERE ct.dcId = 'oldDoctor' and cs.DCID = ct.CONTEXTSPECIFIC_DCID;
```

Listing 34: Query for mapping contextualized concepts

With this query we can build our contextualized concept, the resulting mapping is shown in table 38 an example is shown in table 39.

MDO Syntax	**OWL (Manchester Syntax)**
CREATE DIMENSIONAL CONCEPT *contextualized* FOR *dimension* AT *level* AS $concept_1$@$context_1$ ... $concept_n$@$context_n$	Class:*contextualized* EquivalentTo: *dimension* AND atLevel *levelrange* AND (($contextspecific_1$ and not $subcontext_1$) ... OR ($contextspecific_n$))

Table 38: Contextualized OWL Mapping

MDO Syntax	**OWL (Manchester Syntax)**
create dimensional concept oldDoctorCtxlzd FOR Doctor AT Doctor.doctor AS OldDoctor60@NoeContext, OldDoctor60@OoeContext, OldDoctor62@GmundenContext, OldDoctor65@FreistadtContext, OldDoctor62@doctorContext	Class: Doctor.oldDoctorCtxlzd EquivalentTo: Doctor and atLevel some DoctorFrdoctorTodoctor and ((Doctor.oldDoctorCtxlzd1) OR (Doctor.oldDoctorCtxlzd2) OR (Doctor.oldDoctorCtxlzd3) OR (Doctor.oldDoctorCtxlzd4 AND NOT Doctor.FreistadtContext AND NOT Doctor.GmundenContext) OR (Doctor.oldDoctorCtxlzd5 AND NOT Doctor.doctorContext))
	ctxlzd1: Doctor@drmueller, ctxlzd : doctor@Gumden, ctxlzd3:doctor@Freistadt, ctxlzd4: doctor@Oberösterreich, ctxlzd5 :doctor@Niederösterreich

Table 39: Contextualized mapping example

6.5. Extending the MDO-DWH Mapper with Contextualized Concepts

In this section we will show how we will map contextspecific concepts, contextualized concepts and their contexts to our semantic data warehouse.

6.5.1. Contexts

Node-Concept-Level Concepts In contrast to our OWL reasoning, in the semDWH we have all nodes and their role up hierarchy represented by our _NRR tables as shown in section 5.1. With the node rollup tables we can make a simple mapping, shown in table 40. Table 41 contains an example for the mapping.

MDO Syntax	**Generated SQL (SemDWH)**
create dimensional concept *context* FOR *dimension* AT *level* AS *node*;	Select *d* from *dimension* natural join *dimension_nrr* where lvl = *level* and SN = *node*;
create multidimensional flat concept FOR *ds* AS *point*	select *m* from ds_*ds* natural join rs_*ds* where $dimrole_1$_SN = $node_1$ and ... and $dimrole_n$_SN = $node_n$;

Table 40: Node-Concept-Level and Point-Concept-Granularity SQL mapping

MDO Syntax	**Generated SQL (SemDWH)**
create dimensional concept NoeContext FOR Doctor AT Doctor.doctor AS province.Niederösterreich;	select "doctor" from "D_DOCTOR" NATURAL JOIN "D_DOCTOR_NRR" WHERE "DOCTOR_LVL" = 'doctor' AND Doctor_SN = 'Niederösterreich' ;

Table 41: Node-Concept-Level SQL mapping example

Point-Concept-Granularity Concepts Like NCL concepts, PCG concepts are easy to construct using our _RS rollup space tables mentioned in section 5.1. The mapping is shown in table 40.

As with the OWL mapping also in the data warehouse other concepts may be used to represent the concepts.

6.5.2. Contextspecific Concepts

Contextspecific concepts are unproblematic to implement in the semantic data warehouse. All we need to do is make a natural join of the context and the concept. The result will be all nodes/points that fulfill the requirement of the concept in the context. The mapping to SQL is shown in table 42. Table 43 shows an example, again remember that contextspecific contexts are defined implicitly.

MDO Syntax	Generated SQL (SemDWH)
concept@*context*	select *d* from *concept* natural join *context*;

Table 42: Contextspecific SQL mapping

MDO Syntax	Generated SQL (SemDWH)
oldDoctor60@NoeContext	SELECT "DOCTOR" FROM "DC_OLDDOCTOR60" NATURAL JOIN "DC_NOECONTEXT";

Table 43: Contextspecific concept example

6.5.3. Contextualized Concepts

Contextualized concepts are again mapped very similar to the OWL representation. To solve all the problems already mentioned in section 6.4.3 we make a very similar query but adapted it to get our resulting view. In addition we need the view names of the referenced concepts. The query is shown in listing 35. Be aware that this SQL query constructs other SQL queries. The mapping is shown in table 44 and an example in table 45.

```
select LISTAGG ( '(select␣*␣FROM␣"' || n_concept.SQL_NAME || '"' ||
(select distinct '␣MINUS␣select␣*␣from␣' || n_concept.SQL_NAME  || '␣
    NATURAL␣JOIN␣' || sql_name from MDC_DIRECTSUBSUMPTION,
    SQL_NAME_REGISTRY where sub_mdcID = mdo_Id and mdo_construct = '
    multidimensionalconcept' and
        sub_mdcID in (select MDC_CONTEXTSPECIFICT.CONTEXT_MDCID from
            mdc_contextspecforcontextlzdT
         join MDC_CONTEXTSPECIFICT on MDC_CONTEXTSPECIFICT.mdcid =
             mdc_contextspecforcontextlzdT.CONTEXTSPECIFIC_MDCID
         where mdc_contextspecforcontextlzdT.MDCID = ct.mdcID) start with
              sup_mdcID = cs.CONTEXT_MDCID
      CONNECT BY PRIOR sub_mdcid = sup_mdcid) || '')'' , '' UNION '')
      within group (order by cs.mdcID)
      from MDC_CONTEXTSPECIFICT cs,
      mdc_contextspecforcontextlzdT ct,
      SQL_NAME_REGISTRY n_concept
      WHERE
      n_concept.MDO_ID = cs.MDCID and
      n_concept.MDO_CONSTRUCT = 'multidimensionalconcept' and
      ct.mdcId = 'oldDoctor' and
      cs.MDCID = ct.CONTEXTSPECIFIC_MDCID
```

Listing 35: Contextualized concept mapping query for the semantic data warehouse

6.6. Discussion

Contextualized concepts are a good extension to the already existing concepts as they allow us to define concepts in a more meaningful way. The implementation of con-

MDO Syntax	**Generated SQL (SemDWH)**
CREATE DIMENSIONAL CONCEPT *contextualized* FOR *dimension* AT *level* AS $concept_1$@$context_1$ ... $concept_n$@$context_n$	(select d from $contextspecific_1$ minus $subcontext_1$) union ... union (select d from $contextspecific_n$)

Table 44: Contextualized concept SQL Mapping

MDO Syntax	**Generated SQL (SemDWH)**
create dimensional concept oldDoctorCtxlzd FOR Doctor AT Doctor.doctor AS OldDoctor60@NoeContext, OldDoctor60@OoeContext, OldDoctor62@GmundenContext, OldDoctor65@FreistadtContext, OldDoctor62@doctorContext	select "DOCTOR" FROM "DC_OLDDOCTORCTXLZD1") UNION (select "DOCTOR" FROM "DC_OLDDOCTORCTXLZD2") UNION (select "DOCTOR" FROM "DC_OLDDOCTORCTXLZD3") UNION (select "DOCTOR" FROM "DC_OLDDOCTORCTXLZD4" MINUS select "DOCTOR" from DC_FREISTADTCONTEXT MINUS select "DOCTOR" from DC_GMUNDENCONTEXT) UNION (select "DOCTOR" FROM "DC_OLDDOCTORCTXLZD5" MINUS select "DOCTOR" from DC_DOCTORCONTEXT MINUS select "DOCTOR" from DC_DOCTORCONTEXT)
	ctxlzd1: Doctor@drmueller, ctxlzd : doctor@Gumden, ctxlzd3:doctor@Freistadt, ctxlzd4: doctor@Oberösterreich, ctxlzd5 :doctor@Niederösterreich

Table 45: Contextualized concept SQL Mapping

textspecific and contextualized concepts was done without the basis of research papers and therefore needs further investigation if we maybe missed anything when constructing the mappings. One change in contextualized concepts that was already discussed is that in the future we will not allow contextspecific concepts to be used by more than one contextualized concept.

7. Summary and Outlook

In this thesis we presented an implementation for the reasoning over multi-dimensional ontologies. With the prototype on hand we can now define our own concept types using the MDO-Language without need of in depth knowledge of the underlying data structure. The prototype can reason over the created concepts and return a subsumption hierarchy which helps in organizing the concepts. After defining the concepts we can use them in our semantic data warehouse for ontology driven queries.

Limitations of our current prototype are the lack of multi user support and the performance of our reasoning component. Figure 28 shows that reasoning over multidimensional ontologies is the main performance bottleneck. This problem is independent from our implementation. In the future we should look into speeding up the reasoning process by splitting up multidimensional ontologies into different parts for which the subsumption reasoning can be carried out in isolation.

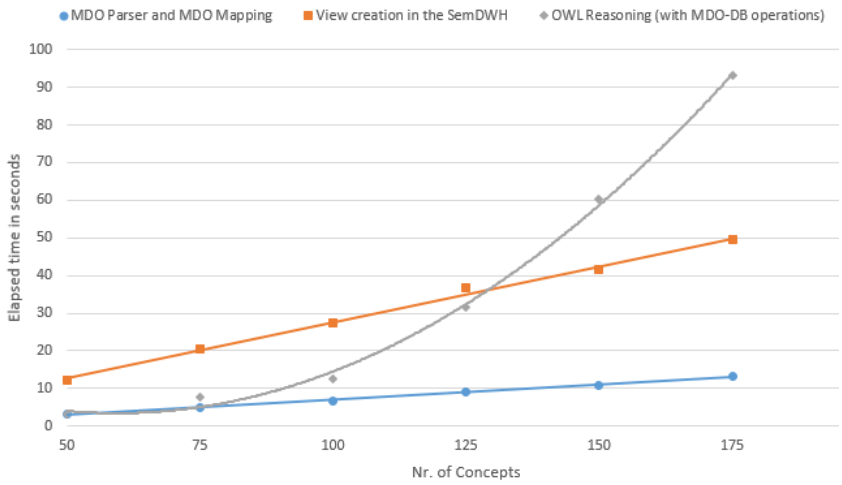

Figure 28: Comparison of the main prototype components

Another problem that might be irritating for the users of this prototype is the need to do many things in triplicate. If we want to use an attribute restricted concept (one of the most natural concept types in the authors opinion) we need to create the entity concept, the corresponding dimensional concept and the corresponding multi-dimensional concept. For further research we should consider the structure being changed to be more intuitive without loosing its explanatory power. This problem may be solved, without the need to change the structure, by a frontend that provides syntactic shortcuts which hide the internal complexities of the approach.

We have shown that reasoning over multi-dimensional is feasible in principle and can be implemented on top of an off-the-shelf database management system and an off-the-shelf OWL reasoner.

References

A. Abello, O. Romero, T. Pedersen, R. Berlanga Llavori, V. Nebot, M. Aramburu, and A. Simitsis. Using semantic web technologies for exploratory olap: A survey. *IEEE Transactions on, Knowledge and Data Engineering*, PP(99):1–1, 2014.

Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. Hermit: an owl 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014.

Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F Patel-Schneider, and Sebastian Rudolph. Owl 2 web ontology language primer. *W3C recommendation*, 2012. URL `http://www.w3.org/TR/owl2-primer/`.

Matthew Horridge and Sean Bechhofer. The owl api: A java api for owl ontologies. *Semantic Web*, 2(1):11–21, 2011.

Matthew Horridge and Peter F Patel-Schneider. Owl 2 web ontology language manchester syntax. *W3C Working Group Note*, December 2009. URL `http://www.w3.org/TR/owl2-manchester-syntax/`.

William H Inmon. The data warehouse and data mining. *Communications of the ACM*, 39(11):49–50, 1996.

Ralph Kimball and Kevin Strehlo. Why decision support fails and how to fix it. *ACM SIGMOD Record*, 24(3):92–97, 1995.

Thomas Neuböck, Bernd Neumayr, Michael Schrefl, and Christoph Schütz. Ontology-driven business intelligence for comparative data analysis. In *Business Intelligence - Proceedings of the Third European Summer School (eBISS 2013), , Vol. 172 of Lecture Notes in Business Information Processing (LNBIP)*, chapter 3, pages 77–120. Springer, 2014.

Bernd Neumayr, Michael Schrefl, and Konrad Linner. Semantic cockpit: an ontology-driven, interactive business intelligence tool for comparative data analysis. In *Advances in Conceptual Modeling - Recent Developments and New Directions, Vol. 6999 of Lecture Notes in Computer Science (LNCS)*, pages 55–64. Springer, 2011.

Bernd Neumayr, Stefan Anderlik, and Michael Schrefl. Towards ontology-based olap: datalog-based reasoning over multidimensional ontologies. In *Proceedings of the fifteenth international workshop on Data warehousing and OLAP*, pages 41–48. ACM, 2012.

Bernd Neumayr, Christoph Schütz, and Michael Schrefl. Semantic enrichment of olap cubes: Multi-dimensional ontologies and their representation in sql and owl. In *On the Move to Meaningful Internet Systems, OTM 2013 Conferences, Vol. 8185 of Lecture Notes in Computer Science*, pages 624–641. Springer, 2013.

Terrence Parr. *The Definitive ANTLR4 Reference*. The Pragmatic Programmers, 2013.

Dieter Steiner. Implementing judgement and analysis rules for comparative data analysis in oracle. Master's thesis, Johannes Kepler Universität Linz, 2014.

Dieter Steiner, Bernd Neumayr, and Michael Schrefl. Judgement and analysis rules for ontology-driven comparative data analysis in data warehouses. In *to appear in: Proceedings of the 11th Asia-Pacific Conference on Conceptual Modelling (APCCM 2015), Sydney, Australia, January 2015, Conferences in Research and Practice in Information Technology (CRPIT)*, 2015.

List of Figures

List of Tables

Listings

A. Running Example in MDO Syntax

Because of compatibility reasons all 'ö' where replaced with 'oe' for this ANTLR commands to work properly the 'ö's have to be changed back again

```
----------------------ENTITY CONCEPTS
create entity concept YoungDoctor for doctor as (doctor.age < '35');
create entity concept VeryYoungDoctor for doctor as (doctor.age < '30');
create entity concept OldDoctor for doctor as (doctor.age > '60');
create entity concept HighPopulationDistrict for district as (district.
    inhabitants > '80000');
create entity concept SmallDistrict for district as (district.sqkm < '150');
create entity Niederoesterreich for province;
create entity concept myProvince for province as {province.Niederoesterreich
    };
create entity concept HighDensDistr for district as ( district.
    HighPopulationDistrict and district.SmallDistrict);
create entity concept YoungOrOldDoc for doctor as ( doctor.OldDoctor or
    doctor.YoungDoctor);
create entity concept notSmallDistrict for district as (NOT district.
    SmallDistrict);

generate dwh objects;
reasoner;

---------------------------DIMENSIONAL CONCEPTS

create dimensional concept InsDuplication for Insurant at Insurant.insurant
    AS sql 'select * from d_insurant';
create dimensional concept byECHighDensDis for Insurant AT Insurant.district
    AS -> (Insurant.district:district.HighDensDistr);
create dimensional concept InHighDensDis for Insurant AT Insurant.insurant ..
     Insurant.district AS expand Insurant.byECHighDensDis;
create dimensional concept InsInHighDensDis FOR Insurant AT Insurant.insurant
     AS Insurant.InHighDensDis Insurant.insurant;
create dimensional concept notHighDensity for Insurant at Insurant.district
    as (NOT Insurant.byECHighDensDis);
create dimensional concept ExpandnotHighDensity for Insurant AT Insurant.
    insurant .. Insurant.district AS expand Insurant.notHighDensity;

create entity concept YoungInsurant for insurant as (insurant.age < '30');
create dimensional concept byECYoungInsurant for Insurant AT Insurant.
    insurant AS -> (Insurant.insurant:insurant.YoungInsurant);
create dimensional concept YoungInsInHighDens for Insurant at Insurant.
    insurant as (Insurant.byECYoungInsurant and Insurant.InsInHighDensDis);
create dimensional concept YoungInsOrHighDens for Insurant at Insurant.
    insurant as (Insurant.byECYoungInsurant or Insurant.InsInHighDensDis);

generate dwh objects;
reasoner;

---------------------------MULTI-DIMENSIONAL CONCEPTS
```

```
create multidimensional flat concept byDcHighDens FOR Ins_Distr AS ->(
    insurant:Insurant.byECHighDensDis);
create unrestricted dimspace InsurantUnrestr With insurant;
create multigranular restricted dimspace Ins_DisIns over InsurantUnrestr At [
    insurant:Insurant.insurant .. Insurant.district];

create multidimensional hierarchic concept expandHighDens FOR Ins_DisIns AS
    expand byDcHighDens;

create monogranular restricted dimspace Ins_Insurants over InsurantUnrestr at
     [insurant:Insurant.insurant];

create multidimensional flat concept restHighDensInsurants FOR Ins_Insurants
    AS expandHighDens [Ins_Insurants];
create multidimensional flat concept notHighDens FOR Ins_Distr AS (not
    byDcHighDens);

create multidimensional flat concept byDcYoungIns FOR Ins_Insurants AS ->(
    insurant:Insurant.byECYoungInsurant);
create multidimensional flat concept YoungInsurantsInHighDensDis FOR
    Ins_Insurants AS (byDcYoungIns and restHighDensInsurants);
create multidimensional flat concept YoungOrHighDensInsurants FOR
    Ins_Insurants AS (byDcYoungIns or restHighDensInsurants);

generate dwh objects;
reasoner;

----------------------------CONTEXTUALIZED CONCEPTS

-Context
create entity Oberoesterreich for province;
create entity Freistadt for district;
create entity Gmunden for district;
create entity Moedling for district;
create entity 24 for doctor;

create entitynode for dimension Doctor entity province.Oberoesterreich at
    Doctor.province;
create entitynode for dimension Doctor entity province.Niederoesterreich at
    Doctor.province;
create entitynode for dimension Doctor entity district.Freistadt at Doctor.
    district;
create entitynode for dimension Doctor entity district.Gmunden at Doctor.
    district;
create entitynode for dimension Doctor entity district.Moedling at Doctor.
    district;
create entitynode for dimension Doctor entity doctor.24 at Doctor.doctor;

create noderollup Doctor.district.Gmunden directlyrollsupto Doctor.province.
    Oberoesterreich;
create noderollup Doctor.district.Freistadt directlyrollsupto Doctor.province
    .Oberoesterreich;
create noderollup Doctor.district.Moedling directlyrollsupto Doctor.province.
    Niederoesterreich;
```

```
create noderollup Doctor.doctor.24 directlyrollsupto Doctor.district.Moedling
    ;

create dimensional concept DocDuplication for Doctor at Doctor.doctor AS sql
    'select * from d_doctor';
create dimensional concept NoeContext FOR Doctor AT Doctor.doctor AS Doctor.
    province.Niederoesterreich Doctor.DocDuplication Doctor.doctor;
create dimensional concept OoeContext FOR Doctor AT Doctor.doctor AS Doctor.
    province.Oberoesterreich Doctor.DocDuplication Doctor.doctor;
create dimensional concept GmundenContext FOR Doctor AT Doctor.doctor AS
    Doctor.district.Gmunden Doctor.DocDuplication Doctor.doctor;
create dimensional concept FreistadtContext FOR Doctor AT Doctor.doctor AS
    Doctor.district.Freistadt Doctor.DocDuplication Doctor.doctor;
create dimensional concept MoedlingContext FOR Doctor AT Doctor.doctor AS
    Doctor.district.Moedling Doctor.DocDuplication Doctor.doctor;
create dimensional concept doctorContext FOR Doctor AT Doctor.doctor AS
    Doctor.doctor.24 Doctor.DocDuplication Doctor.doctor;

generate dwh objects;
reasoner;

create entity concept OldDoctor60 for doctor as (doctor.age > '59');
create entity concept OldDoctor62 for doctor as (doctor.age > '62');
create entity concept OldDoctor65 for doctor as (doctor.age > '65');

create dimensional concept OldDoctor60 for Doctor AT Doctor.doctor AS -> (
    Doctor.doctor:doctor.OldDoctor60);
create dimensional concept OldDoctor62 for Doctor AT Doctor.doctor AS -> (
    Doctor.doctor:doctor.OldDoctor62);
create dimensional concept OldDoctor65 for Doctor AT Doctor.doctor AS -> (
    Doctor.doctor:doctor.OldDoctor65);

generate dwh objects;
reasoner;


-DC Contextualized Concept
create dimensional concept oldDoctorCtxlzd FOR Doctor AT Doctor.doctor AS
    Doctor.OldDoctor60@Doctor.NoeContext, Doctor.OldDoctor60@Doctor.
    OoeContext, Doctor.OldDoctor62@Doctor.GmundenContext, Doctor.
    OldDoctor65@Doctor.FreistadtContext, Doctor.OldDoctor62@Doctor.
    doctorContext;

generate dwh objects;
reasoner;
```

Listing 36: ANTLR inserts

B. ANTLR Grammar of the MDO Syntax

```
grammar Mdo;
/*
@header {
package at.jku.dke.semcockpit.mdo.repl.parser.antlr;
}
*/
/**
*** ------------------------------------------------------------------------
*** Parser Start Rule
*** (incl. misc. helpers)
*** ------------------------------------------------------------------------
**/
mdo
    :   ((mdo_dl | mdo_ql | mdo_query) SEMICOLON)+;

mdo_dl
    :  CREATE mdo_dl_options mdo_dl_constructs;

mdo_dl_options
    :   orReplace? mdoOnly? sqlName?;

mdo_dl_constructs
    :   dl_create_entityClass
    |   dl_create_entity
    |   dl_create_conventionalDimension
    |   dl_create_semanticDimension
    |   dl_create_dimrole_onDimension
    |   dl_create_unrestricted_dimspace
    |   dl_create_restricted_dimspace
    |   dl_create_universalDimRoleSpace
    |   dl_create_dimRoleSpace
    |   dl_create_factClass
    |   dl_create_entityConcept
    |   dl_create_dimConcept
    |   dl_create_mdConcept
    |   dl_create_entitynode
    |   dl_create_nodeRollup
    |   dl_create_measure
    |   dl_create_score
    ;

mdo_ql
    :   APPLY mdo_ql_applications
    ;

mdo_ql_applications
    :   ql_measureApplication
    |   ql_scoreApplication
    ;
```

```
mdo_query
    :   (SHOW mdo_constructs)
    |   generate_dwh
    ;

mdo_constructs
    :   ENTITYCLASSES           #showEntityclass
    |   ENTITYCONCEPTS          #showEntityConcepts
    |   ATTRIBUTES              #showAttributes
    |   DIMENSIONS              #showDimensions
    |   LEVELS                  #showLevels
    |   (DIMCONCEPTS|DIMENSIONALCONCEPTS)             #showDimconcepts
    |   DIMROLES                #showDimroles
    |   (RESTRICTED DIMSPACES)  #showRestrictedDimspaces
    |   (UNRESTRICTED DIMSPACES)    #showUnrestrictedDimspaces
    |   (MDCONCEPTS|MULTIDIMENSIONALCONCEPTS)   #showMdconcepts
    ;

generate_dwh
    :   GENERATE DWH OBJECTS
    ;

/**
*** ---------------------------------------------------------------
*** Top-Level Parser Rules
*** ---------------------------------------------------------------
**/

dl_create_entityClass
    :   ENTITY CLASS
            entityClassName LPAREN
                entityClassAttributes
                orderByEntityClassAttributes
            RPAREN
    ;

dl_create_entity
    :   ENTITY entityName FOR entityClassName;

dl_create_conventionalDimension
    :   CONVENTIONAL DIMENSION conventionalDimensionName LPAREN
            levelName
            (
                COMMA
                levelName
            )*
        RPAREN;

dl_create_semanticDimension
    :   SEMANTIC DIMENSION semanticDimensionName
            FROM externalOntology
            ROOTED IN owlConcept
    ;
```

```
dl_create_dimrole_onDimension
    : DIMROLE dimRoleName ON DIMENSION dimensionName;

dl_create_unrestricted_dimspace
    : UNRESTRICTED DIMSPACE dimSpaceName WITH dimRoleName (COMMA dimRoleName)
       *;

dl_create_restricted_dimspace
    : MONOGRANULAR RESTRICTED DIMSPACE monoDimspaceDefinition
    | MULTIGRANULAR RESTRICTED DIMSPACE multiDimspaceDefinition;

monoDimspaceDefinition
    : dimSpaceName OVER dimSpaceName AT granularityDefInline;

multiDimspaceDefinition
    : dimSpaceName OVER dimSpaceName AT granularityDefInline;

dl_create_universalDimRoleSpace
    :   UNIVERSAL DIMROLESPACE LPAREN
            dimRoleDef (COMMA dimRoleDef)*
            (CONSTRAINTS equiConstraint (COMMA equiConstraint)* )?
        RPAREN;

dl_create_dimRoleSpace
    :   DIMROLESPACE dimRoleSpaceName dimRoleSpaceDef;

dl_create_factClass
    :   FACT CLASS factClassName
        FOR dimRoleSpaceName AT granularityDefInline
        LPAREN baseMeasureDef (COMMA baseMeasureDef) * RPAREN
        LPAREN
            COMPLETE FOR dimRoleConceptName (COMMA dimRoleConceptName)*
        RPAREN
    ;

dl_create_entityConcept
    :   ENTITY CONCEPT
                entityConceptName
            FOR entityClassName
            AS  ceDescription
    ;

dl_create_dimConcept
    :   DIMENSIONAL CONCEPT dimConceptName
        FOR dimensionName AT levelRange AS cdDescription
    ;

dl_create_mdConcept
    :   MULTIDIMENSIONAL (HIERARCHIC|FLAT) CONCEPT mdConceptName
        FOR dimSpaceName AS cmDescription
    ;

dl_create_entitynode
```

```
    :   ENTITYNODE FOR DIMENSION dimensionName ENTITY entityName AT levelName
        ;

dl_create_nodeRollup
    :   NODEROLLUP nodeName DIRECTLYROLLSUPTO nodeName;

dl_create_measure //"derived" measure
    :   MEASURE derivedmeasureName
        (LPAREN formalQualifierName (COMMA formalQualifierName) RPAREN)?
        (DATATYPE datatype)?
        OVER factClassName (COMMA factClassName)
        (FOR (INTERSECTION_SPACE | UNION_SPACE | dimRoleSpaceName))?
        (AT (COMMON_ROLLUP_GRANULARITIES | granularityRange))?
        AS
        measurementInstruction
    ;

dl_create_score
    :   SCORE scoreName
        (LPAREN formalQualifierName (COMMA formalQualifierName) RPAREN)?
        (DATATYPE datatype)?
        OVER factClassName (COMMA factClassName)
        (FOR (INTERSECTION_SPACE | UNION_SPACE | dimRoleSpaceName))?
        (AT LPAREN (COMMON_ROLLUP_GRANULARITIES | granularityRange) RPAREN
            COMMA
            LPAREN (COMMON_ROLLUP_GRANULARITIES | granularityRange) RPAREN)?
        AS
        comparisonInstruction
    ;

ql_measureApplication
    :   MEASURE mdQuery;

ql_scoreApplication
    :   SCORE comparativeMdQuery;

/**
***
    ------------------------------------------------------------------------

*** Misc. Parser Rules
***
    ------------------------------------------------------------------------

**/

orReplace
    :   (OR REPLACE);

sqlName
    :   (SQL_NAME constant);

mdoOnly
    :   MDO_ONLY;
```

```
entityClassAttributeWithId
    :   attribute ID;

orderByEntityClassAttributes
    :   (| ORDER BY attributeNameWithSortOrder
           (COMMA attributeNameWithSortOrder)* );

attributeNameWithSortOrder
    :   attributeName sortOrder?;

sortOrder
    :   (ASC | DESC);

entityClassAttributes
    :   (attribute | entityClassAttributeWithId)
        (COMMA (attribute | entityClassAttributeWithId))*;

measure_qualified
    :   measureName
        LPAREN actualQualifier (COMMA actualQualifier) RPAREN
    ;

score_qualified
    :   scoreName
        LPAREN actualQualifier (COMMA actualQualifier) RPAREN
    ;

default_ //default was?
    :   constant | NULL;

actualQualifier
    :   (qualifierName COLON)? (
            multiDimConceptName
        |   formalQualifierName
        )
    ;

comparisonInstruction
    :   SQL sqlQuery
    |   aggregationComparison
    |   arithmeticComparison
    ;

aggregationComparison
    :   LPAREN
            granularityRange DBL_DOT measure_qualified COMMA
            granularityRange DBL_DOT measure_qualified
        RPAREN
    ;

arithmeticComparison
    :   LPAREN measure_qualified COMMA measure_qualified RPAREN
    ;
```

```
measurementInstruction
    :   SQL sqlQuery
    |   calcTerm
    ;

calcTerm
    :   simpleTerm
    |   LPAREN complexTerm RPAREN;

simpleTerm
    :   UNARY_ARITHMETIC_OP? (
                constant
            |   ql_measureApplication
            |   ql_scoreApplication
            |   aggregExpr
        )
        default_?
    ;

aggregExpr
    :   AGGREGATION_OP LPAREN mdQuery | comparativeMdQuery RPAREN;

mdQuery
    :   pointSliceGranularityFragment DBL_DOT measureName COALESCE;

comparativeMdQuery
    :   pointSliceGranularityFragment COMMA
        (scoreJoinCondition)* COMMA
        pointSliceGranularityFragment
        DBL_DOT measureName;

scoreJoinCondition
    :   scoreJoinConditionParticipant
        EQUALS (NEXT | PREVIOUS)?
        scoreJoinConditionParticipant
    ;

scoreJoinConditionParticipant
    :   (GOC | GOI) DOT dimRoleName DOT levelName;

pointSliceGranularityFragment
    :   point? externalSliceCondition? granularityRange?;

externalSliceCondition
    :   multiDimConceptName;

complexTerm
    :   calcTerm BINARY_ARITHMETIC_OP calcTerm;

levelRange
    :   (levelName | BOTTOM | TOP | ALL) (DBL_DOT (levelName | TOP | ALL))?;

granularityRange_singleRole
```

```
    :   dimRoleName COLON levelRange;

granularityRange
    :   granularityRange_singleRole (COMMA granularityRange_singleRole)*;

granularityDefInline
    :   LPAREN_SQUARE
            dimRoleName COLON levelRange (COMMA dimRoleName COLON levelRange)
                *
        RPAREN_SQUARE
    ;

baseMeasureDef
    :   baseMeasureName baseMeasureDatatype;

dimRoleSpaceDef
    :   dimRoleName (COMMA dimRoleName)* ;

dimRoleDef
    :  dimRoleName
        REFERENCES dimensionName
    ;

equiConstraint
    :   dimRoleName COLON levelName
        EQUALS
        dimRoleName COLON levelName
    ;

dimensionLevelName
    :   dimensionName
        COLON
        levelName
        SEMICOLON
    ;

attribute
    :  attributeName datatype;

ceDescription
    :   PRIMITIVE               #primitiveEc
    |   (SQL sqlQuery )         #sqlEc
    |   ceExpr                  #definedEc
    ;

cdDescription
    :   PRIMITIVE               #primitiveDc
    |   (SQL sqlQuery )         #sqlDc
    |   cdExpr                  #definedDc
    |   nodeConceptLevel        #nodeConceptLevelDc
    |   conceptLevel            #conceptLevelDc
    ;

cmDescription
```

```
    :   PRIMITIVE                   #primitiveMdc
    |   (SQL sqlQuery )            #sqlMdc
    |   cmFactbased                #factbasedMdc
    |   pointConceptGranularity    #pointConceptGranMdc
    |   DIMSPACECONCEPT            #dimspaceMdc
    |   cmExpr                     #definedMdc
    ;

cmFactbased
    :   (ql_measureApplication | ql_scoreApplication)
        comparisonOp
        (ql_measureApplication | ql_scoreApplication | constant);

pointConceptGranularity
    :   point? mdConceptName LPAREN_SQUARE dimSpaceName RPAREN_SQUARE
    ;
//granularityRange substituted by dimspace name

point
    :   COMPARISON_OP (
            (dimRoleName COLON nodeName (COMMA dimRoleName COLON nodeName)*)
        |   SELF )
        COMPARISON_OP;
//used comparison_op '<' and '>' are defined in more than one rule and
   otherwise it is not possible

ceExpr
    :   entityConceptName       #conceptName
    |   ceByAttribute           #ecByAttribute
    |   ceDisjunctiveOp         #ecDisjunctive
    |   ceConjunctiveOp         #ecConjunctive
    |   ceBinaryOp              #ecBinaryOperator
    |   ceComplement            #ecComplement
    |   ceNominals              #ecNominal
    ;

cdExpr
    :   (
                dimConceptName
            |   cdByEntityConceptName
            |   cdDisjunctiveOp
            |   cdConjunctiveOp
            |   cdBinaryOp
            |   cdComplement
            |   cdHierarchyExpansion
            |   cdContextualized
            //|   owlConceptName
            //|   owlConceptExpr
        )
        (STAR)?
    ;

cmExpr
    :   (
```

```
                mdConceptName
            |   cmByDimensionalConcept
            |   cmDisjunctiveOp
            |   cmConjunctiveOp
            |   cmBinaryOp
            |   cmComplement
            |   cmHierarchyExpansion
            |   cmNominals
            |   cmContextualized
        )
        (STAR)?
    ;

nodeConceptLevel
    :       nodeName dimConceptName levelRange;

conceptLevel
    :       dimConceptName levelRange;

ceByAttribute
    :   LPAREN attributeName comparisonOp constant RPAREN;

cdByEntityConceptName
    :   R_ARROW LPAREN levelName COLON entityConceptName RPAREN;

cdHierarchyExpansion
    :   EXPAND dimConceptName;

ceDisjunctiveOp
    :   LPAREN ceExpr (OR ceExpr)* RPAREN;

ceConjunctiveOp
    :   LPAREN ceExpr (AND ceExpr)* RPAREN;

ceBinaryOp
    :   LPAREN ceExpr ( AND | OR ) ceExpr RPAREN;

ceComplement
    :   LPAREN NOT ceExpr RPAREN;

ceNominals
    :   LPAREN_CURLY entityName (COMMA entityName)* RPAREN_CURLY;

cdDisjunctiveOp
    :   LPAREN cdExpr (OR cdExpr)* RPAREN;

cdConjunctiveOp
    :   LPAREN cdExpr (AND cdExpr)* RPAREN;

cdBinaryOp
    :   LPAREN cdExpr ( AND | OR ) cdExpr RPAREN;

cdComplement
    :   LPAREN NOT cdExpr RPAREN;
```

```
cdContextualized
    : dimConceptName AT_SIGN context (COMMA dimConceptName AT_SIGN context)*;

context
    : dimConceptName;

/*cdNominals //??
    :   LPAREN_CURLY dimensionName (COMMA dimensionName)* RPAREN_CURLY;*/

cmContextualized
    : mdConceptName AT_SIGN mdContext (COMMA mdConceptName AT_SIGN mdContext)
       *;

mdContext
    : mdConceptName;

cmDisjunctiveOp
    :   LPAREN cmExpr (OR cmExpr)* RPAREN;

cmConjunctiveOp
    :   LPAREN cmExpr (AND cmExpr)* RPAREN;

cmBinaryOp
    :   LPAREN cmExpr ( AND | OR ) cmExpr RPAREN;

cmComplement
    :   LPAREN NOT cmExpr RPAREN;

cmHierarchyExpansion
    :   EXPAND mdConceptName;

cmNominals
    :   LPAREN_CURLY point (COMMA point)* RPAREN_CURLY;

cmByDimensionalConcept
    :   R_ARROW LPAREN dimRoleName COLON dimConceptName RPAREN;

/**
***
    ----------------------------------------------------------------------------

*** Identifiers
***
    ----------------------------------------------------------------------------

**/

measureName:        baseMeasureName | derivedmeasureName;
qualifierName:      genericIdentifier;
scoreName:          genericIdentifier;
nodeName:           genericIdentifier;
entityClassName:    genericIdentifier;
conventionalDimensionName
```

```
    :               genericIdentifier;
semanticDimensionName
    :               genericIdentifier;
entityName:          genericIdentifier;
attributeName:      genericIdentifier;
entityConceptName:  genericIdentifier;
dimConceptName:     genericIdentifier;
multiDimConceptName:genericIdentifier;
cmByDimRoleConceptName
    :               genericIdentifier;
mdConceptName:      genericIdentifier;
dimRoleConceptName: genericIdentifier;
dimensionName:      genericIdentifier;
levelName:          genericIdentifier;
dimRoleName:        genericIdentifier;
externalOntology:   genericIdentifier;
owlConcept:         genericIdentifier;
dimRoleSpaceName:   genericIdentifier;
dimSpaceName:       genericIdentifier;
factClassName:      genericIdentifier;
baseMeasureName:    genericIdentifier;
derivedmeasureName: genericIdentifier;
formalQualifierName:genericIdentifier;

/**
***
   ------------------------------------------------------------------------------

*** Basic Parser Rules
***
   ------------------------------------------------------------------------------

**/
comparisonOp:   COMPARISON_OP;

constant:       String_Literal;

sqlQuery:       String_Literal;

genericIdentifier
        :       Ident;

datatype:       FLOAT | INTEGER | VARCHAR | DATE;

baseMeasureDatatype
    :           FLOAT | INTEGER;

/**
***
   ------------------------------------------------------------------------------

*** Tokens
***
   ------------------------------------------------------------------------------
```

```
**/

CREATE:         C R E A T E;
REPLACE:        R E P L A C E;
ENTITY:         E N T I T Y;
CLASS:          C L A S S;
ID:             I D;
ORDER:          O R D E R;
BY:             B Y;
CONCEPT:        C O N C E P T;
FOR:            F O R;
AS:             A S;
PRIMITIVE:      P R I M I T I V E;
SQL:            S Q L;
NOT:            N O T;
OR:             O R;
AND:            A N D;
FLOAT:          F L O A T;
INTEGER:        I N T E G E R;
STRING:         S T R I N G;
DATE:           D A T E;
VARCHAR:        V A R C H A R;
CONVENTIONAL:   C O N V E N T I O N A L;
SEMANTIC:       S E M A N T I C;
DIMENSION:      D I M E N S I O N;
DIMENSIONS:     D I M E N S I O N S;
DIMROLE:        D I M R O L E;
DIMROLES:       D I M R O L E S;
HIERARCHY:      H I E R A R C H Y;
HIERARCHIC:     H I E R A R C H I C;
DIMSPACECONCEPT: D I M S P A C E C O N C E P T;
FLAT:           F L A T;
LEVEL:          L E V E L;
LEVELS:         L E V E L S;
FROM:           F R O M;
ROOTED:         R O O T E D;
EXPAND:         E X P A N D;
IN:             I N;
ON:             O N;
UNRESTRICTED:   U N R E S T R I C T E D;
MONOGRANULAR:   M O N O G R A N U L A R;
MULTIGRANULAR:  M U L T I G R A N U L A R;
RESTRICTED:     R E S T R I C T E D;
DIMSPACE:       D I M S P A C E;
DIMSPACES:      D I M S P A C E S;
UNIVERSAL:      U N I V E R S A L;
DIMROLESPACE:   D I M R O L E SPACE;
REFERENCES:     R E F E R E N C E S;
CONSTRAINTS:    C O N S T R A I N T S;
FACT:           F A C T;
AT:             A T;
COMPLETE:       C O M P L E T E;
BOTTOM:         B O T T O M;
```

```
TOP:            T O P;
ALL:            A L L;
DIMENSIONAL:    D I M E N S I O N A L;
MULTIDIMENSIONAL
    :           M U L T I D I M E N S I O N A L;
MDCONCEPTS:     M D C O N C E P T S;
MULTIDIMENSIONALCONCEPTS
    :           M U L T I D I M E N S I O N A L C O N C E P T S;
SELF:           S E L F;
MEASURE:        M E A S U R E;
DATATYPE:       D A T A T Y P E;
OVER:           O V E R;
WITH:           W I T H;
INTERSECTION_SPACE
    :           I N T E R S E C T I O N UNDERSCORE SPACE;
UNION_SPACE
    :           U N I O N UNDERSCORE SPACE;
COMMON_ROLLUP_GRANULARITIES
    :           C O M M O N UNDERSCORE R O L L U P UNDERSCORE
                G R A N U L A R I T I E S;
//DEFAULT:        D E F A U L T;
COALESCE:       C O A L E S C E;
NULL:           N U L L;
GOI:            G O I;
GOC:            G O C;
NEXT:           N E X T;
PREVIOUS:       P R E V I O U S;
SCORE:          S C O R E;
APPLY:          A P P L Y;
SHOW:           S H O W;
ENTITYCLASSES:  E N T I T Y C L A S S E S;
ENTITYNODE:     E N T I T Y N O D E;
ATTRIBUTES:     A T T R I B U T E S;
ENTITYCONCEPTS: E N T I T Y C O N C E P T S;
GENERATE:       G E N E R A T E;
DWH:            D W H;
OBJECTS:        O B J E C T S;
DIMCONCEPTS:    D I M C O N C E P T S;
DIMENSIONALCONCEPTS:
                D I M E N S I O N A L C O N C E P T S;
NODEROLLUP:     N O D E R O L L U P;
DIRECTLYROLLSUPTO: D I R E C T L Y R O L L S U P T O;

MDO_ONLY:       M D O UNDERSCORE O N L Y;
SQL_NAME:       S Q L UNDERSCORE N A M E;

COMPARISON_OP:  '<' | '>=' | '>' |'=<' | EQUALS_FRG;

LPAREN_ANGLE:   '<';
RPAREN_ANGLE:   '>';
LPAREN:         '(';
RPAREN:         ')';
LPAREN_CURLY:   '{';
RPAREN_CURLY:   '}';
```

```
LPAREN_SQUARE:  '[';
RPAREN_SQUARE:  ']';
SEMICOLON:      ';';
COLON:          ':';
DOT:            '.';
DBL_DOT:        '..';
STAR:           '*';
AT_SIGN:        '@';
COMMA:          ',';

R_ARROW:        '->';

EQUALS:         EQUALS_FRG;

AGGREGATION_OP: SUM | MIN | MAX | AVG | COUNT;

BINARY_ARITHMETIC_OP
    :           '+' | '-' | '*' | '/';

UNARY_ARITHMETIC_OP
    :           '-';

AGGREGATION_COMPARISON_OP
    :           MEAN_PERCENTILE_RANK | MEDIAN_PERCENTILE_RANK;

ARITHMETIC_COMPARISON_OP
    :           PERCENTAGE_DIFFERENCE | RATIO;

fragment MEAN_PERCENTILE_RANK
    :           M E A N UNDERSCORE P E R C E N T I L E UNDERSCORE R A N K;

fragment MEDIAN_PERCENTILE_RANK
    :           M E D I A N UNDERSCORE P E R C E N T I L E UNDERSCORE R A N K
       ;

fragment PERCENTAGE_DIFFERENCE
    :           P E R C E N T A G E UNDERSCORE D I F F E R E N C E;

fragment RATIO
    :           R A T I O;

fragment SUM:   S U M;
fragment MIN:   M I N;
fragment MAX:   M A X;
fragment AVG:   A V G;
fragment COUNT: C O U N T;
fragment SPACE: S P A C E;
fragment UNDERSCORE: '_';
fragment EQUALS_FRG:    '=';

ASC:            A S C;
DESC:           D E S C;

// Case-insensitive alpha characters
```

```
fragment A: ('a'|'A');
fragment B: ('b'|'B');
fragment C: ('c'|'C');
fragment D: ('d'|'D');
fragment E: ('e'|'E');
fragment F: ('f'|'F');
fragment G: ('g'|'G');
fragment H: ('h'|'H');
fragment I: ('i'|'I');
fragment J: ('j'|'J');
fragment K: ('k'|'K');
fragment L: ('l'|'L');
fragment M: ('m'|'M');
fragment N: ('n'|'N');
fragment O: ('o'|'O');
fragment P: ('p'|'P');
fragment Q: ('q'|'Q');
fragment R: ('r'|'R');
fragment S: ('s'|'S');
fragment T: ('t'|'T');
fragment U: ('u'|'U');
fragment V: ('v'|'V');
fragment W: ('w'|'W');
fragment X: ('x'|'X');
fragment Y: ('y'|'Y');
fragment Z: ('z'|'Z');

String_Literal
    : '\'' (~('\'') | '\'' '\'' )* '\''
    ;

Quoted_Name
    : '\"' (~('\"') | '\"' '\"')* '\"'
    ;

fragment Digit
    : '0'..'9'
    ;

fragment Letter
    : ('A'..'Z' | 'a'..'z' | 'oe' | 'ae' | 'ue' | 'OE' | 'AE' | 'UE' | 'SZ' )
    ;

fragment Hex
    : ('A'..'F' | 'a'..'f' | '0'..'9')
    ;

Ident
    : (Letter | Digit)(Letter | Digit | '_' | '.')*
    ;

Integer
    : '-'? Digit+
    ;
```

```
Qmark
    : '?'
    ;

/*
 * Normally a lexer only emits one token at a time, but ours is tricked out
 * to support multiple (see @lexer::members near the top of the grammar).
 */
Float
    : Integer '.' Digit*
    ;

Ws
    : (' ' | '\t' | '\n' | '\r')+ -> channel(HIDDEN)
    ;

Comment
    : ('--' | '//') .*? ('\n'|'\r') -> channel(HIDDEN)
    ;
```

Listing 37: ANTLR Grammar