JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Simon Schausberger, BSc

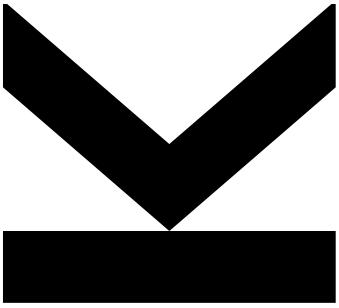
Submitted at
Department of Business Informatics – Data & Knowledge Engineering

Supervisor
o. Univ.-Prof. DI Dr. Michael Schrefl

Co-Supervisor
Ass.-Prof. Mag. Dr. Christoph Schütz

November 2016

The Semantic Data Warehouse for the AgriProKnow Project
A First Prototype



Master Thesis
to obtain the academic degree of
Master of Science
in the Master's Program
Business Informatics

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at

Eidesstattliche Erklärung

Ich, Simon Schausberger, 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.

Signed: _____

Date: November 4, 2016

Acknowledgements

I am indebted to many people for their support during the years of my studies. Without them the successful completion of my master thesis would not have been possible. In the following paragraphs I would like to thank some of them. I am aware that there are many more and nothing I write here can express the gratitude I feel for them.

I am grateful to all members of the department for Data & Knowledge Engineering, for supporting me and for providing a very productive working environment. Especially I would like to thank o. Univ.-Prof. DI Dr. Michael Schrefl who gave valuable feedback regarding my thesis and Ass.-Prof. Mag. Dr. Christoph Schütz for providing useful information, comments, remarks, and help whenever needed. Moreover, I would like to thank Dr. Schütz for his guidance in writing scientific reports and his incredible support in the last steps of creating this thesis.

Furthermore, I would like to thank my family, particularly my parents, which supported me in every life decision I ever made and who taught me so much. Last but not least I would like to thank my girlfriend Katja, who supported me in unimaginable ways. She was always successful in making me smile, even on the most exhausting days.

Abstract

Contemporary dairy farming heavily relies on modern technology such as milking robots, feeding systems, and various sensors which track animal movement, micro climate, etc. All these systems produce vast amounts of data. These data contain potentially valuable information that could be used to increase efficiency of dairy farm operations. As of now this potential remains underused, which the AgriProKnow project intends to change. The AgriProKnow project develops a data analysis platform as a means to extract knowledge from the information contained in the data. In this thesis we present a first prototype of the AgriProKnow project's data analysis platform in the form of a semantic data warehouse (sDWH).

The sDWH is realised using a combination of semantic technologies and a relational database management system. The schema and all instance data are described in RDF format using the RDF Data Cube Vocabulary. The RDF schema is mapped to a relational data model; the instance data in the sDHW are stored in a relational database. Furthermore, the sDWH provides intuitive query facilities for the stored data, the semOLAP patterns. The semOLAP patterns are defined by database and domain experts. Each semOLAP pattern contains wildcards. Based on the semOLAP patterns, users create queries and provide concrete values for the wildcards in the pattern. The combination of the semOLAP pattern and concrete values for its wildcards results in an SQL query which is executed in the relational database of the sDWH. If the concrete values for the wildcards of the semOLAP pattern are RDF elements, the export of the query results can be done in RDF as well. The query result is enriched semantically including, a definition of the result's structure and the underlying query.

Zusamenfassung

Moderne Rinderbetriebe setzen eine Vielzahl moderner Technologien wie etwa Temperatursensoren, Milchroboter oder Ftterungsroboter ein. Dadurch entstehen große Mengen an Daten in den Betrieben. Diese Daten enthalten potenziell wertvolle Informationen, welche zur Effizienzsteigerung genutzt werden können. Das Potenzial dieser Informationen wird derzeit jedoch nicht voll ausgeschöpft. Das Projekt AgriProKnow will dies ändern. Deshalb wird im Rahmen des Projekts AgriProKnow eine Analyseplattform entwickelt, die helfen soll, neues Wissen aus in den Daten enthaltenen Information zu generieren. In dieser Arbeit wird ein erster Prototyp dieser Analyseplattform, dem Semantic Data Warehouse (sDWH) beschrieben.

Das sDWH kombiniert semantische Technologien mit einem relationalen Datenbankverwaltungssystem. Die Schema- und Instanzdaten des sDWH werden in RDF mittels RDF Data Cube Vocabulary beschrieben. Das RDF-Schema wird in einem relationalen Datenmodell abgebildet. Die Instanzdaten werden in einer relationalen Datenbank gespeichert. Das sDHW bietet außerdem intuitive Abfragemöglichkeiten, die semOLAP Patterns. Diese werden von Datenbank- und Domänenexperten definiert, und beinhalten Platzhalter. Erstellen die Nutzer Abfragen werden diese Platzhalter mit konkreten Werten gefüllt. Basierend auf diesen Werten und dem zugrundeliegenden Pattern, wird daraus eine SQL-Abfrage generiert, welche in der relationalen Datenbank des sDWH ausgeführt wird. Wurden die Werte für das Pattern in Form von RDF-Elementen angegeben, erfolgt auch der Export im RDF-Format. Hierbei wird zusätzlich zu den Daten des Ergebnisses, eine semantische Beschreibung in Form einer Definition der Struktur des Ergebnisses und der zugrundeliegenden Abfrage exportiert.

Contents

1 Introduction

The vast amounts of data generated in modern dairy farming contain potentially valuable information that can be used to increase efficiency of dairy farm operations. As of now this potential remains underused, which the AgriProKnow project intends to change. The AgriProKnow project develops a data analysis platform as a means to extract knowledge from the information contained in the data, which may facilitate the work of dairy farm managers, farming consultants, equipment vendors and veterinarians. In this thesis we present a first prototype of the AgriProKnow project's data analysis platform in the form of a semantic data warehouse.

1.1 The AgriProKnow Project

The AgriProKnow project is a joint research effort under the lead of Smartbow GmbH[1], between the Department for Business Informatics – Data & Knowledge Engineering[2] and the Institute for Stochastics[3] of Johannes Kepler University Linz, the Department for Farm Animals and Veterinary Public Health of the University of Veterinary Medicine Vienna[4], Josephinum Research from the Federal Institute of Education and Research (HBLFA) Francisco Josephinum[5], and Wasserbauer GmbH[6]. The project aims at creating novel information-based methods to increase efficiency of milk production in precision dairy farming.

Contemporary dairy farming heavily relies on modern technology such as milking robots, feeding systems, and various sensors which track animal movement, micro climate, etc. All these systems produce vast amounts of data in different formats, typically existing in isolation. In order to leverage the information contained in the produced data the AgriProKnow project taps into the various data sources that exist in modern dairy farms. To this end the AgriProKnow project builds on the results of AgriOpenLink [24] which is a data integration platform that employs semantic technologies to overcome the differences in data format of the various sources. In AgriProKnow we transform these data into a form fit for analysis and provide intuitive query facilities for end users.

At the core of the AgriProKnow project is a *semantic data warehouse* (sDWH), a first prototype of which we present in this thesis. The sDWH contains integrated data from various sources described using semantic technologies. The data are organised following the multidimensional model, consisting of cubes and dimensions, realised in this first prototype using a relational database management system. The description of the semantics of the data in the sDWH allows to link the data to existing domain ontologies.

The AgriProKnow project uses the farm operation data for data mining, analytical queries and rule-based farm operations. Prior to loading the data into the sDWH, data mining uncovers knowledge about the dairy farming process which can be used for analytical queries in the sDWH. Furthermore, statistical methods are used to preprocess the raw source data for noise reduction and preaggregation. Dairy farm managers, farming consultants, equipment vendors and veterinarians

[1] http://www.smartbow.at/
[2] http://www.dke.jku.at/
[3] http://www.jku.at/stochastik/
[4] http://www.vetmeduni.ac.at/universitaet/departments/department3/
[5] http://www.josephinum.at/en/blt/josephinum-research.html
[6] http://www.wasserbauer.at/

may then run analytical queries on the data in the sDWH. These users are hardly experts in database technology. Thus, the sDWH must provide intuitive query facilities. Finally, analytical rules periodically scan the data in the sDWH for events of interest and trigger corresponding actions. For example, a risk for disease discovered in the data automatically results in calling a veterinarian.

In the AgriProKnow project each partner is responsible for a specific part of the AgriProKnow project. Smartbow as project leader preprocesses sensor data. Josephinum Research is responsible for ontology design and modelling of process knowledge. The Insitute of Stochastics deals with data mining and the development of statistical models. Wasserbauer GmbH consults on automated feeding systems. The Departement of Business Informatics – Data & Knowledge Engineering develops the sDWH. In this thesis we present a first version of data model and loading infrastructure for the AgriProKnow project's sDWH as well as intuitive query facilities for analysts. We do not cover data mining and analytical rules.

1.2 The Semantic Data Warehouse

The AgriProKnow project's sDWH should provide historical data for statistical analysis at an adequate level of detail. In order to guarantee sensible results as far as possible the sDWH must take into account the differences in quality of the underlying data and cope with incomplete data. Furthermore, the sDWH must be open to future additions to the data model. The sDWH should also provide facilities for intuitive querying and ease interpretation of the results.

Concerning the choice of database management system (DBMS) the first prototype opts for a relational DBMS. Since the sDWH should store historical data, a stream DBMS was discarded although a stream DBMS may serve for data mining and preaggregation of the source data. With SQL as a powerful query language relational DBMS provide rich capabilities for statistical analysis, in particular when proprietary extensions are factored in. Relational DBMS represents mature and reliable technology that has proven to be capable of handling large amounts of data. Although relational DBMS often reach their limits in big data applications such as processing of sensor data, in the case of the AgriProKnow project the preprocessing of data reduces the size of the data in such way that the data can be handled by relational DBMS. Yet, the choice of relational DBMS is not the rationale behind preaggregation. Rather, storing the full amount of sensor data at highly detailed levels of granularity is expensive and provides no gain for the analysis. The detailed data is noisy and more abstract data is sufficient for generating knowledge.

The AgriProKnow project is an ongoing endeavour, its data model is subject to constant revision and extension. Moreover, once operative, the AgriProKnow data analysis platform should be open to agricultural domains apart from dairy farming, e.g., pig farming. Thus, the sDWH provides possibilities for schema creation and modification, which gives allows to evolve the data model during the project and beyond, without the need to apply tedious changes to the source code directly.

The sDWH provides concise representation of all data related to a specific subject in subject-oriented views. The subject-oriented views are created through combination of sDWH cubes with the same subject. For example, cubes containing milking data from different sources such as milking robots or official authorities are combined in a subject-oriented view about milk. Since the source cubes may differ in quality and frequency of the recorded data subject-oriented views

fill gaps with continuation. For example, official authorities track milk quality only once a month whereas milking robots constantly analyse milk contents. Still, the subject-oriented view may contain data from both robots and authorities on a daily basis. The same milk quality data from authorities, however, is used continuously for days between points of measurement, with indication of freshness of the data. Continuation avoids null values which are hard to handle during analysis.

Recurring patterns of analysis can be formalised using semOLAP patterns. Database and business intelligence experts in collaboration with domain experts formulate semOLAP patterns using SQL and the semOLAP pattern definition language. Possible patterns are, for example, kinds of comparison such as the comparison of measures for different groups. Analysts may then instantiate semOLAP patterns by providing concrete values for wildcard elements within the pattern definition. For example, an analyst may compare milk output from two different groups of animals.

Using an RDF vocabulary for data cubes, the sDWH provides machine-readable representation of its cubes which can be linked to existing ontologies in the agricultural domain, as semantic technologies and ontologies are used within the agricultural domain [5, 13, 20]. Future work will also use semantic technologies to describe query output.

1.3 Outline

The remainder of this thesis is organised as follows. Chapter 2 (Background) briefly introduces the technologies and concepts that were used to construct the first prototype of the sDWH as well as possible alternatives. The background describes work on RDF in conjunction with relational databases and work on intuitive query facilities. Chapter 3 (System Overview) gives an overall view of the sDWH architecture. Chapter 4 (Schema Definition and Data Loading) presents the sDWH interface for creating and adapting the schema as well as loading the data. Chapter 5 (Data Analysis) presents the analysis view and introduces semOLAP patterns. The thesis concludes with a summary and outlook on future work.

2 Background

In this chapter we briefly present existing technologies that served to implement the sDWH of the AgriProKnow project. In particular, we focus on two research topics. First, we introduce technologies that combine RDF and relational databases; these technologies are the fundamental of the sDWH. Second, we present related work on intuitive querying.

2.1 RDF and Relational Databases

The main data source for the AgriProKnow project is AgriOpenLink [24], which uses RDF as data format. The data within the AgriProKnow sDWH are stored in a relational database (RDB). Thus, some kind of mapping between these technologies is necessary. The World Wide Web Consortium (W3C) has two different recommendations for mapping data from a relational database to RDF. Bumans [4] and Michel et al.[15] describe further possibilities for a mapping from a relational data model to RDF.

The first recommendation for a mapping from relational data to RDF is the direct mapping of the data [1]. The result of the mapping is the *direct graph* which is the union of the multiple *table graphs*. Each table graph contains the data of one table in the RDB. Each row of the table is represented as *row graph*. Direct mapping allows to convert relational tables into RDF graphs, but the direct mapping does not specify a mapping vocabulary. The direct mapping consists of a set of rules to transform a relational database into an RDF graph.

Another option for a mapping from relational data to RDF, also a recommendation by the W3C, is the RDB to RDF Mapping Language (R2RML) [7]. In R2RML, *triples maps* are used to map each table into RDF. Each triples map defines the mapping of one table. A triples map can be seen as set of rules to convert relational data into RDF triples.

The two recommendations made by the W3C focus on relational databases, but in AgriProKnow we use a relational database to store multidimensional data. Therefore, we do not focus on the relational model *per se* but on cubes, dimensions, etc. The Multidimensional to RDF Mapping Language (M2RML) [11], which is based on R2RML, provides a mapping from multidimensional concepts to RDF. The M2RML mapping approach employs the RDF Data Cube Vocabulary [6], combining M2RML with the RDF data cube vocabulary, effectively transforming relational data that represents multidimensional data into RDF data.

With regards to the AgriProKnow project, the problem of the described mapping possibilities is their focus on the conversion from relational to RDF data, which is sufficient for the export of query results in RDF. As our data source is in RDF format however, we load these data into a relational database and, therefore, also need a solution which transforms RDF data into relational data.

We focus on the description of multidimensional schema in RDF and develop an application for the transformation of relational data into RDF. We employ the RDF Data Cube Vocabulary (qb) [6], which is also used by M2RML, for the definition of multidimensional schemas. The qb vocabulary serves for schema and data description in RDF. As qb has some limitations, e.g., the inability to define dimension hierarchies, we further employ its extension qb for OLAP (qb4o) [2] to overcome

the limitations of qb. A more detailed description of qb and its extension qb4o follow in the next sections.

2.1.1 The RDF Data Cube Vocabulary

The RDF Data Cube Vocabulary (qb) is a W3C recommendation which is used to publish multi-dimensional data with a focus on statistical data. The qb vocabulary is based on the SDMX (Statistical Data and Metadata eXchange) ISO standard [7]. The information in this section originates from the official W3C recommendation [6]. In this section an overview of qb is given with focus on the features used in this thesis. Qb provides further possibilities to structure and publish data which are not used in this thesis and therefore not discussed.

The main concept of qb are cubes. Each cube contains data. Data within a cube is represented as `qb:DataSet`. A `qb:DataSet` is referenced by `qb:Observations`. Each `qb:Observation` is one entry in the cube and follows the cube's structure, which is referenced using the `qb:structure` property by the `qb:DataSet`. The cube's structure is defined as `qb:DataStructureDefinition` and may contain dimensions, measures and attributes. Defining the cube structure has several advantages. As the structure is defined the structure of the data can be checked against the structure definition to assure a complete DataSet. Moreover, it is easier to get an overview of the data as it is stated which dimensions, measures and attributes are present. Dimensions, measures and attributes are added to a `qb:DataStructureDefinition` using the `qb:component` property, as they are all subproperties of the abstract `qb:Component` class. Listing 1 shows an example data set about blood measurements with an example observation of a particular measurement of the calcium level in a cow's blood.

Listing 1: Example of a qb:DataSet and qb:Observation

```
agrid:DataSet_BloodMeasurement_Calcium_1 a qb:DataSet;
    qb:structure  agri:BloodMeasurement_Calcium .

agrid:BloodMeasurement_Calcium_1 a qb:Observation;
    qb:dataSet agrid:DataSet_BloodMeasurement_Calcium_1
    agri:Animal agrid:Kuh_1 ;
    agri:Calcium 23.7;
    agri:Date_ agrid:2016-04-17.
```

A referenced measure is created using the `qb:MeasureProperty` type. Within the definition of a measure the type of its values should be stated by using the `rdfs:range` property and an according XML type, e.g., `xsd:string`, `xsd:double` or `xsd:integer` (see Listing 2). The definition of attributes follows the same principle as the definition of measures, except that attributes are of type `qb:AttributeProperty`. When referencing a measure or attribute additional properties can be used, e.g., to order components, which are not used within the thesis and therefore not described further.

Listing 2: Example of a qb:MeasureProperty and qb:AttributeProperty

```
agri:Calcium a qb:MeasureProperty;
```

[7] http://www.iso.org/iso/catalogue_detail.htm?csnumber=52500

```
    rdfs:range xsd:double.

agri:EventName a qb:AttributeProperty;
    rdfs:range xsd:string.
```

In addition to measures and attributes an important part of a cube are dimensions. Dimensions are defined using the `qb:DimensionProperty` type. All values of a dimension are predefined in code lists. There are two main types of code lists, namely hierarchical and non-hierarchical code lists. An example for a non-hierarchical code list would be the gender dimension, this dimension may only have the values male, female or other. These three values do not have any hierarchical connection. A hierarchical dimension would be, e.g., an origin dimension. Values of an origin dimension might be Europe and Austria. These two values are in a hierarchical relation, as Austria is a country in Europe and therefore a narrower description of origin. To create code lists for dimensions either the SKOS vocaublary (Simple Knowledge Organization System) [16] or the `qb:HierarchialCodeList` property are the only options mandated by the qb recommendation. Dimensions as defined in the qb vocabulary are restricted and only provide basic hierarchies, which are not suitable for the AgriProKnow sDWH and OLAP (Online Analytical Processing) in general. Therefore, the QB for OLAP vocabulary was created which is described in the next section.

2.1.2 QB for OLAP

QB for OLAP (qb4o) is an RDF vocabulary developed by Etcheverry et al. ([2], [9]). We use version 1.3 of the qb4o vocabulary. The main goal of qb4o is the extension of the qb vocabulary in order to make qb suitable for OLAP. Therefore, it is possible to use existing qb structures and extend them with qb4o elements. As qb lacks support regarding dimension hierarchies, qb4o is mostly providing vocabulary to create dimension hierarchies. The qb4o vocabulary provides possibilities to define dimension hierarchies and instances of these dimension hierarchies. Another benefit of qb4o lies in the definition of aggregation functions, but as those are solved within the AgriProKnow project as a separate concept, namely *calculated measures*, aggregation functions are not described further here.

Qb4o organises dimension structures around levels and hierarchies. Each `qb:DimensionProperty` represents a dimension and has to be associated with at least one `qb4o:Hierarchy`, which in turn has to consist of at least one `qb4o:LevelProperty`. Each `qb4o:LevelProperty` must belong to at least one hierarchy. A level has at least one `qb:AttributeProperty` referenced by `qb4o:hasId`. All of the properties referenced by `qb4o:hasId` combined are the primary key of a dimension level. In addition to the primary key attributes a level may contain `qb4o:LevelAttributes`, these are referenced using the `qb4o:hasAttribute` property and are only describing attributes of a level. To order the levels within a hierarchy `qb4o:HierarchySteps` are defined. Every `qb4o:HierarchyStep` states a child and a parent level as well as the cardinality between those levels and a roll-up property.

Listing 3 shows the definition of the animal dimension, which consists of the levels, animal, main breed and date of birth. Two hierarchies are defined within the animal dimension. The first hierarchy is defined with animal as child and main breed parent level. The second also has animal as child level and defines date of birth as animal's parent level.

Listing 3: Example for a Dimension in qb4o

```
## Dimension definition
agri:AnimalDim a qb:DimensionProperty;
    qb4o:hasHierarchy agri:AnimalHier_MainBreed, agri:
       AnimalHier_Dob.

## Dimension hierarchies
agri:AnimalHier_MainBreed a qb4o:Hierarchy;
    qb4o:hasLevel agri:Animal, agri:MainBreed.

agri:AnimalHier_Dob a qb4o:Hierarchy;
    qb4o:hasLevel agri:Animal, agri:DateOfBirth.

# Dimension Levels
agri:Animal a qb4o:LevelProperty;
    qb4o:hasID agri:NationalID;
    qb4o:hasAttribute agri:AnimalName.

agri:MainBreed a qb4o:LevelProperty;
    qb4o:hasID agri:Breed.

agri:DateOfBirth a qb4o:LevelProperty;
    qb4o:hasID agri:Date_Val.

## ID attributes of dimension levels
agri:NationalID a qb:AttributeProperty;
    rdfs:range xsd:string;
    xsd:maxLength 14.

agri:Breed a qb:AttributeProperty;
    rdfs:range xsd:string.

agri:Date_Val a qb:AttributeProperty;
    rdfs:range xsd:date.

## Describing attribute
agri:AnimalName a qb4o:LevelAttribute;
    rdfs:range xsd:string.

## Rollups
agri:hasMainBreed a qb4o:RollupProperty.
agri:hasDateOfBirth a qb4o:RollupProperty.

## hierarchy steps
_:AnimalHier_MainBreed_hs1 a qb4o:HierarchyStep;
    qb4o:inHierarchy agri:AnimalHier_MainBreed;
    qb4o:childLevel agri:Animal;
    qb4o:parentLevel agri:MainBreed;
    qb4o:pcCardinality qb4o:ManyToOne;
```

```
    qb4o:rollup agri:hasMainBreed.

_:AnimalHier_Dob_hs1 a qb4o:HierarchyStep;
    qb4o:inHierarchy agri:AnimalHier_Dob;
    qb4o:childLevel agri:Animal;
    qb4o:parentLevel agri:DateOfBirth;
    qb4o:pcCardinality qb4o:ManyToOne;
    qb4o:rollup agri:hasDateOfBirth.
```

After the definition of the dimension structure, instance data for the dimension level are defined. An instance within a dimension level is of type `qb4o:LevelMember` and is associated to the specific level using the `qb4o:memberOf` property. The relation between two instances of different levels is stated using the roll-up property defined in the according hierarchy step. Listing 4 shows an example, which defines a member of level animal rolling up to a member of level main breed.

Listing 4: Example for Instance Data of a qb4o Dimension

```
agrid:Kuh_1 a qb4o:LevelMember ;
    qb4o:memberOf agri:Animal
    agri:AnimalName "NAGERL" ;
    agri:NationalID  "AT000182633814" ;
    agri:hasDateOfBirth agrid:2006_12_26;
    agri:hasMainBreed agrid:Holstein-Schwarzbunt.

agrid:Holstein-Schwarzbunt a qb4o:LevelMember ;
    qb4o:memberOf agri:MainBreed;
    agri:Breed "Holstein-Schwarzbunt".


agrid:2006_12_26 a qb4o:LevelMember.
    qb4o:memberOf  agri:DateOfBirth;
    agri:Date_Val  "2006-12-26".
```

The defined dimensions are referenced within a cube. Therefore, qb4o extends `qb:DataStructureDefinition`. Using qb, the `qb:DimensionProperty` is referenced from the `qb:DataStructureDefinition`, which has to be extended. In qb4o the dimension is organised in levels, a level of the dimension has to be referenced from the `qb:DataStructureDefinition` using the `qb4o:level` property. Furthermore, a cardinality between fact and dimension must be specified with an instance of `qb4o:Cardinality`, which traditionally and most commonly are many-to-one (many facts to one dimension member). Listing 5 shows the definition of a blood measurement cube for calcium measurements. The cube references the dimension levels animal and date, and includes a measure for calcium-level in an animal's blood.

Listing 5: Example of a DataStructureDefinition using qb and qb4o

```
agri:BloodMeasurement_Calcium a qb:DataStructureDefinition;
    qb:component
        [qb4o:level agri:Date;
        qb4o:cardinality qb4o:ManyToOne];
    qb:component
        [qb4o:level agri:Animal;
```

```
        qb4o:cardinality qb4o:ManyToOne];
    qb:component [qb:measure agri:Calcium].
```

Qb4o alone may not serve for the generation of an SQL schema. Therefore, in this thesis, a further extension of qb/qb4o was created which will be described in sect ion 4.2. The description of qb4o concludes the subsection on RDF vocabularies.

2.2 Intuitive Query Facilities

Another main challenge of the AgriProKnow project is to provide functionalities for querying of the sDWH, which can be used by non-experts in database technology. In the following we present different approaches to analyse the data through queries which does not need extensive knowledge of a query language as described in literature.

One approach to simplify querying are Visual Query Systems (VQS) [3, 8]. VQS are based on query-by-example and provide user-friendly query interfaces. For usability reasons VQS are often implemented as web-based systems. With these systems users are able to create queries through the use of forms and form elements, such form elements being buttons, drop down menus, etc. Sometimes special forms are predefined to ease the creation of common queries within a system. VQS, however, focus on the user interface, they simplify how a query can be created, but the users still have to have knowledge about all possible operations and how they work if it comes to complex queries. Therefore, even if an interface with high usability exists, it might be hard for non-expert users to create more complex queries.

Another more formal approach would be the use of multidimensional algebra [19]. It was proven that only a subset of functionalities provided by relational algebra is used, if data is queried from an analytical point of view [14]. Therefore multidimensional algebra was developed to express analytical queries in a more effective way. Within the AgriProKnow project querying is used to analyse data in a multidimensional model. Multidimensional algebra focuses on expressing such queries by providing a set of operators for multidimensional models. Although there are only seven operators within multidimensional algebra it is still necessary for the user to learn the algebra to express their queries and gain knowledge of multidimensional models, which might be a complex task for users without knowledge of database systems.

Based on multidimensional algebra, Varga et. al [25] developed a tool for OLAP operations on data in qb/qb4o format called QB2OLAP. Varga et. al developed a way to query RDF data without the need of any SPARQL or RDF knowledge. To do so Varga et. al created the language QL which implements operations of multidimensional algebra for qb/qb4o data. As the defined operations are only for data in RDF format, this approach cannot be used within the AgriProKnow project, as our data is stored within a relational database.

An approach which relies on SQL but has got a higher-level view of querying is BIRD by Schuetz et. al [22]. Schuetz et. al describe analysis situations, where an analysis situation is defined as a specific view of an analyst on the data to solve a task. As an analysis situation contains parameters, it is possible to reuse an analysis situation in different tasks by changing the situation parameters. Therefore, an analysis situation can be seen as a generic query. To specify a specific query the analysis situation's parameters are filled with specific values. Among the possible parameter types are predicates and calculated measures, which are both defined having a name and an expression.

Predicates define business terms, which after definition can be used within analysis situations. Benefits of these global definitions are that the analyst only needs to know the name and meaning of the term but there is no need to know the term's exact definition and even though the analyst might not know the exact definition an analyst always uses the exact same expression when using a term. The same applies for calculated measures, which describe key performance indicators and are derived from other measures.

Based on the analysis situation, Schuetz et. al [22] describe analysis graphs, originally developed in the semCockpit project [17], that guide the users through the data. An analysis graph are analysis situations which are connected through OLAP operations. While working with one analysis situation the user is provided with possible OLAP operations, which can be applied to the situation and lead to a new analysis situation. A related form of navigation is implemented in the herd management system DairyComp[8], which shows the user related queries based on operational data.

The solution implemented in the AgriProKnow project is influenced by the approaches above, mainly by BIRD and its analysis situations. Moreover, the idea of predicates and calculated measures is taken from BIRD. As our data model is multidimensional, the idea of multidimensional algebra is also part of our solution.

[8] http://web.vas.com/en/Products/Detail/4570

3 System Overview

Figure 1 illustrates the architecture of the sDWH. Sensor data and external data sources such as authority databases are preprocessed and loaded into a relational database; preprocessing of source data is not part of this thesis. Instance data and schema information about the multidimensional model are uploaded to the sDWH using a REST interface. The sDWH consists of a relational database which contains instance data which are used for analysis. A triple store serves as data dictionary that describes the multidimensional model that underlies the relational data. The triple store also serves as staging area for the uploaded instance data which are represented using RDF. Database and business intelligence experts in collaboration with domain experts upload semOLAP pattern definitions into the sDWH, which stores the semOLAP patterns in the triple store. Analysts formulate pattern-based queries which are then executed on the relational database. The sDWH returns semantically enriched query results in RDF format.

3.1 Schema Definition and Data Loading

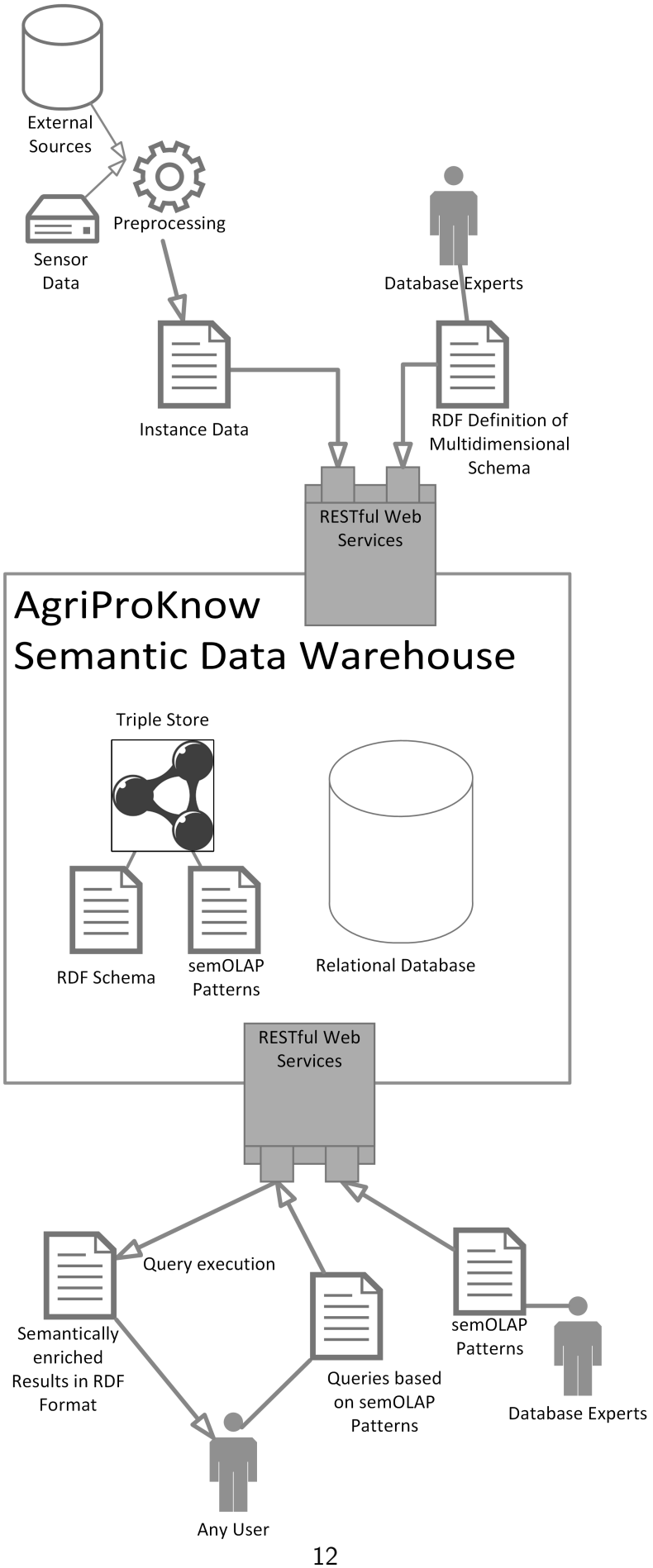
Since the AgriProKnow project is work-in-progress the multidimensional model of the sDWH is constantly evolving. Modifications of the multidimensional model should be possible without direct access to the source code by database experts. Therefore, database experts define the sDWH schema using the RDF Data Cube Vocabulary (qb), a W3C recommendation [6], and its extension qb for OLAP (qb4o) [10]. The qbgen vocabulary provides further extensions of qb and qb4o specifically designed in order to realise the sDWH features required for the AgriProKnow project. Although designed for the AgriProKnow project the qbgen vocabulary may be used for other sDWH projects with similar requirements. In particular, the qbgen vocabulary introduces complex attributes which provide additional information about one or more dimensions.

Loading schema definition and instance data into the sDWH must follow a well-defined loading procedure. This loading procedure consists of a series of RESTful web service calls. First, a loading request must be issued. loading requests include the data that should be loaded into the sDWH. All the schema information and instance data are defined using the previously mentioned RDF vocabularies. The loading request also explicitly states whether it concerns schema information or instance data, and whether the loading request is an insert, replace or delete. The sDWH stores an issued loading request in chronological order. Issuing an execution command for schema information or instance data loading requests results in the execution of the previously issued loading requests for schema information or instance data, respectively.

A loading request for schema information may be either creation of cubes, dimension or complex attributes, addition of dimension attributes or deletion of cubes. With a relational implementation the creation of schema elements corresponds to the creation of tables. Based on the sent RDF data the sDWH creates a number of tables. The RDF data remains in the triple store as metadata that describes the created tables. The creation of the relational schema from the RDF data follows a specific convention, basically a star schema.

loading request for instance data may be insert/replace or delete. Instance data is also uploaded as RDF data which is then translated into a series of SQL insert, update or delete statements. Future work may improve performance of the loading process by uploading CSV files which can be loaded into the database more efficiently.

Figure 1: System Architecture of the Semantic Data Warehouse



In the AgriProKnow project, data originates from various sensors and databases, e.g., ear tags, temperatures sensors, milking or feeding robots, official authorities or dairies. Typically, the data in their raw format are unsuitable for analysis and must be preprocessed accordingly before being loaded into the sDWH. Preprocessing consists of abstraction, harmonization, semantic differentiation and restructuring.

Sensors typically track data at high levels of granularity, e.g., movement sensors carry out ten readings per second. High levels of granularity lead to huge storage space requirements and potential runtime issues for the analysis. Yet, the high levels of granularity generated by sensors are usually not needed for analysis of historical data. Prior to loading the data, abstraction reduces the level of detail, making the amount of data more manageable.

Different sources employ different units of measurement for the same measures. In order to be comparable the units of measurement of the source data must be harmonized.

Different sources measure the same kinds of facts, e.g., both milking parlours and dairies record milk quantity. Quality of the data may differ between sources. Thus, the sDWH should differentiate between data sources when recording measures. Consequently, analysts may use this provenance information in their analyses.

Restructuring transforms the source data into RDF data that corresponds to the multidimensional model defined using the qb, qb4o and qbgen vocabularies. The loaded dataset must provide values for all fields, i.e., null values are not allowed.

The RDF data for an instance data loading request are staged in the triple store. SPARQL queries transform the data into SQL insert, update, delete statements. After execution of the SQL statements the RDF instance data are removed from the triple store.

3.2 Data Analysis

Subject-oriented views provide a more integrated view on a particular subject. Subject-oriented views are created by combination and completion of the loaded instance data. Each base cube in the sDWH addresses a specific subject as recorded by a particular data source. Subject-oriented views combine data from different cubes about the same subject across different data sources. Furthermore, data sources track data at different frequencies. For example, official authorities track milk quality only once a month whereas milking robots constantly analyse milk contents. A subject-oriented view that combines data from these sources completes the missing authority data for days without measurement in order to avoid null values. Completion is done by continuation of the last available measurement. Continuation extends for a specified number of days. If the available measurement is too old, no continuation is applied and a null value persists. The schema also indicates the freshness of a continued measurement.

Another feature of subject-oriented views is addition of information. In the specific case of AgriProKnow some subject-oriented views contain calving information. To this end a subject-oriented view is combined with data from the calving cube, thus indicating the time since the last calving (DayOfLactation) and the number of the last calving. Calving information is often used in queries.

The properties of subject-oriented views are specified using a RESTful web service. An analysis view generator creates the subject-oriented views according to the configuration. The analysis

view generator is implemented as PL/SQL stored procedures in the database. The analysis view generator composes the subject-oriented view definition by combining SQL snippets. The subject-oriented view definition is an SQL query that is used to create a materialised view.

The definition of semOLAP patterns facilitates querying of the analysis view. A semOLAP pattern describes the structure of a query for a specific purpose, e.g., a certain kind of comparison. A semOLAP pattern consists of several elements with well-defined semantics. A pattern expression defines the translation of a semOLAP pattern into a target language; in the case of AgriProKnow, the target language is SQL. pattern expressions are specified in a domain-specific language. The pattern expression language is based on the target language, e.g., SQL. A pattern expression can be seen as an SQL query which includes wildcards. Wildcards correspond to the pattern elements. The pattern definition also contains definitions of the range of the pattern elements. pattern elements can be reused in multiple patterns. Finally, a semOLAP pattern defines its output in terms of pattern elements.

Analysts may instantiate semOLAP patterns by providing concrete values for the pattern elements. Patterns can be instantiated on different levels of abstraction, namely ROLAP and qb level. If instantiated at the ROLAP level analysts provide names of tables and columns representing facts, dimensions, levels and attributes of the multidimensional model. If instantiated at the qb level analysts provide qb, qb4o and qbgen elements. Furthermore, analysts may instantiate pattern elements with calculated measures and predicates. Calculated measures and predicates are defined by domain and database experts in order to unambiguously express the calculation of key performance indicators and the semantics of business terms for reuse in multiple queries. Both predicates and calculated measures are based on existing schema elements. The definitions of predicates and calculated measures consist of the used schema elements and include an expression in the target language that is used for query generation. For ROLAP applications the expression must be in SQL and the schema elements are columns.

The definition of semOLAP patterns, calculated measures and predicates as well as pattern instances is represented in RDF. A RESTful web service allows users to upload these definitions into the sDWH. The definitions are stored in the triple store of the sDWH. An uploaded pattern instance may then be executed multiple times using a separate web service.

A ROLAP pattern instance translates into an SQL query. Translation is done using the pattern expression. The translation of pattern instances at the qb level require a mapping from qb, qb4o and qbgen elements to relational model elements. Execution of a ROLAP pattern instance results in a CSV file whereas execution of a qb pattern instance results in a qb data structure. The advantage of qb pattern instances lies in the semantic description of the query result in terms of the multidimensional model which facilitates the interpretation of the result.

4 Schema Definition and Data Loading

The first part of this thesis revolves around the initial creation of the semantic data warehouse (sDWH). As the multidimensional model is developed within the AgriProKnow project, there was no predefined model to begin with. At the start the database and the triple store are blank without any schema or data. Therefore, the initial development included two tasks. The development of interfaces for schema creation and modification and, the development of interfaces to load data.

In compliance with the goals of AgriProKnow both schema data and instance data are defined in RDF format using the already discussed qb and qb4o vocabularies. As both schema and instances are defined in the same way, their loading process into the sDWH is similar and follows the procedure discussed in Section 4.1.

The differences in schema creation/modification and loading of data are the web service endpoints used. Furthermore, the definition of schema and instance data use different parts of the qb/qb4o vocabularies and the definition of instance data references the defined schema. Therefore, and as the database is without schema in the beginning, the schema of the sDWH has to be created at first. After an initial schema has been created it is possible to load data into the schema. As the multidimensional model within AgriProKnow is continually evolving it is possible to modify the schema, without the loss of already loaded instance data.

The following sections will take a closer look at the general loading procedure, the creation and modification of the sDWH schema and the loading of data into the sDWH.

4.1 Loading Procedure

Schema and instance data are both defined in RDF format and loaded into the sDWH using web services. Therefore they follow the same loading procedure, although they use different web service endpoints. All data to be loaded into the sDWH have to be present as RDF in Turtle syntax and have to be described using the qb/qb4o vocabularies and their extension qbgen. The Turtle syntax was chosen because of its readability and lesser overhead compared to RDF/XML. RDF data in other syntactical formats can be converted to Turtle using the Jena framework[9]. For communication with the sDWH RESTful webservices are provided. The loading of schema and instance data is done in two steps. First, a loading request with the data has to be issued. Secondly, the loading request has to be executed. After execution the error log can be retrieved.

4.1.1 Loading Requests

The first, operation is the issuing of a loading request. It provides the sDWH with the neccesary informations for schema creation/modification and loading of data. A loading request is issued through a HTTP POST request with MIME type text/plain. All data have to be included as the body of the HTTP POST request in RDF/Turtle format using qb, qb4o and qbgen vocabularies.

A loading request either includes schema data or instance data. Depending on the type of data there are a number of web service endpoints used to issue the loading requests, which are described

[9] `https://jena.apache.org/`

in a later section. The web services which are used to issue loading requests return HTTP status codes as a result. If status code 202 (accepted) was returned, the operation was successful and the loading request has been queued. If status code 500 (server error) is returned, there was an error during the operation and further details can be seen in the error log (Section 4.1.3).

When a loading request is issued it does not take immediate effect within the sDWH. Instead the data sent is stored as file and an entry in the table `SchemaLoadRequest` or `InstanceLoadRequest` of the database, depending on the type of loading request, is created. Both of these tables are used to queue loading requests for later execution. This approach was chosen as there is no need for real time data analysis in the AgriProKnow project. With this approach data can be uploaded into the sDWH at any time without using much computing resources and disrupting ongoing analysis. If the system's workload is low the loading of schema and instance data can be started, without disrupting other ongoing operations.

4.1.2 Execution Commands

After loading requests are loaded into the sDWH, execution is started using execution commands. As two types of data exist (schema and instance data) there are two according types of execution commands, `ExecuteSchemaChanges` and `ExecuteInstanceChanges`. To issue an execution command, a specific type has to be chosen. The execution command is then issued through a HTTP GET request with MIME type text/plain to the URL `<server>/AgriProKnowDBService/rest/ExecutionService/ExecuteSchemaChanges` to execute schema changes and `<server>/AgriProKnowDBService/rest/ExecutionService/ExecuteInstanceChanges` to execute changes of instance data.

The execution command returns a HTTP status code as result. Code 202 (accepted) is returned, if the execution of loading requests was successfully started. Code 409 (conflict) is returned, if an execution is already running, as only one running execution command is allowed at a time. Code 500 (server error) is returned, if an error occurred during the start of execution and an entry in the error log is created as well.

After an execution command has been received by the sDWH, processing of the according loading requests is started asynchronously. Requests are loaded from the queue starting with the oldest request. Only one request is processed at a time. The file containing the loading request data is loaded into a separate named graph of the triple store. Each is processed in isolation to the others.

If a request fails, an entry in the error log is created and execution continues with the next one. If the loading requests that fails contains instance data, the database is rolled back, which removes all changes made by the loading request. If the loading request includes schema data and fails with a `SchemaExecutionExceptions` the sDWH schema should be checked, as schema creation is done by DDL statements which cannot be rolled back and may have already taken effect on the sDWH schema. If the loading request with schema data fails with another exception no DDL statements have been executed on the database yet.

If a major error, e.g., loss of database connection, occurs the execution stops and all loading requests that have not been fully processed remain in queue. Furthermore, an entry in the error log is created. The outcome of the execution command does not have any effect on the returned

HTTP status code, as the execution happens asynchronously to the execution command. Problems during execution only appear in the the error log, which is described in the following section.

4.1.3 Error Log

As the execution of loading requests happens asynchronously, users can't be given immediate feedback about any errors occurring. Therefore the sDWH uses an error log to store all occurred errors and the data detailing them. The error log can be retrieved using a HTTP GET request on the URL `<server>/AgriProKnowDBService/rest/ExecutionService/GetAllErrors`. Additionally, the parameter `deleteErrors=true` can be added to clear the error log after retrieval.

The error log is returned as CSV using the semicolon (;) as field separator. As there are three main types of errors, the structure of each line varies depending on the main type of error it describes. The three types are, schema request errors, instance request errors and application errors. Schema and instance request errors are directly connected to a loading request. In contrast to an application error which cannot be linked to a loading request. Schema and instance request errors are only caused by a predefined set of exceptions which are described below, after a detailed description of the main error types and their structure. In the future the error log might be presented in RDF format which would add new possibilities for error description, e.g., to reference or include a faulty RDF definition.

Schema Request Errors. A schema request error is directly connected to the execution of a loading request containing schema data. The structure of an according entry in the error log can be seen in Table 1.

Table 1: Error Log Structure for Schema Request Errors

Column	**Description**
log date	Date and time the error occurred and was logged
main error type	Always "SCHEMA_LOAD_REQUEST_"
error name	The specific name of the occurred error. (see Table 4)
message	The error message stating the cause of the error
data file	A link to the file containing the data that caused the error
loaded date	The date and time the loading request was issued
command	The command of the underlying schema load request (see section 4.3)

Instance Request Errors. An instance request error is directly connected to a loading request containing instance data. The structure of an according entry in the error log can be seen in Table 2.

Application Errors. Application esrrors are errors which cannot be directly connected to a loading request. Application errors are severe and stop execution. Examples for application errors

Table 2: Error Log Structure for Instance Request Errors

Column	**Description**
log date	Date and time the error occurred and was logged
main error type	Always "INSTANCE_LOAD_REQUEST"
error name	The specific name of the occurred error. (see Table 4)
message	The error message stating the cause of the error
data file	A link to the file containing the data that caused the error
loaded date	The date and time the loading request was issued
command	The command of the underlying instance load request (see section 4.4)
parameter	The base table parameters value of the loading request

Table 3: Error Log Structure for Application Errors

Column	**Description**
log date	Date and time the error occurred and was logged
main error type	Always APPLICATION_ERROR
error name	The specific name of the occurred error. In case of an application error any Java exception.
message	The error message stating the cause of the error; including a limited stack trace
source	Description during which operation the error occurred

are: Loss of connection to the database/triple store; missing configuration files, etc. The structure of an according entry in the error log can be seen in Table 3.

Causes of Schema and Instance Request Errors. Schema and instance request errors can only be caused by a defined number of exceptions. All those exceptions are stated and described in Table 4. These exceptions only occur during execution of loading requests.

The loading procedure which was described here is used while loading schema and instance data. Another thing common between schema and instance data is the use of qbgen vocabulary which is described in the next section.

Table 4: Exceptions during Execution of a Loading Request

Exception	**Description**
NotFoundInSchemaException	A stated name could not be found in the RDF schema
QueryException	An exception during execution of a query occurred.
FileTroubleException	An exception during the processing of a file occurred.
NoKeysException	A table to be generated does not have any key declared in the RDF schema.
NoLevelsException	A dimension does not have levels.
FaultyHierarchyException	A dimension hierarchy is faulty, e.g., has multiple or no root levels.
SchemaScriptExecutionException	During the execution of script, to create or change the database schema, an error occurred. The relational chema should be checked in the database, as some changes have already taken effect.
NameToLongException	A column or table name used is too long for the database.
NoColumnsException	A table to be generated does not have any columns specified in the RDF schema.
ComplexAttributeNotFoundException	A stated complex attribute could not be found in the RDF schema.
DegenerateDimensionException	It was tried to perform a command on a degenerate dimension not suitable for it.
FactNotFoundException	A stated fact could not be found in the RDF schema.
GroupRelationDataNotFoundException	Data regarding a relation of two tables could not be found.
NoDataDeletedException	No data for deletion could be found in the file provided for deletion.
NoDataInsertedException	No data for insertion could be found in the file provided for insertion.
NoSnowflakeDimensionException	It was tried to perform a command only suitable for snowflake dimensions on a non-snowflake dimension.
NoSnowflakeLevelException	It was tried to perform a command only suitable for levels of snowflake dimensions on a level not part of a snowflake dimension.

4.2 The qbgen Vocabulary as Extension of qb and qb4o

Qb and qb4o are the basic vocabularies for defining the sDWH schema, but they do not meet the full requirements for the AgriProKnow project's sDWH. On the one hand, qb and qb4o do not provide all information needed for generation of the schema. On the other hand, there are additional requirements which are not fulfilled by qb and qb4o. Therefore, the vocabulary qb for generation (qbgen) was developed as part of this thesis. Qbgen's goals are similar to the ones of qb4o. It is only an extension of the existing vocabularies qb and qb4o. A schema already defined in qb/qb4o can be taken and extended with qbgen, without losing any information. Qbgen's main goal is the addition of information for SQL schema generation, as some information is missing in the qb/qb4o vocabularies. Qb and qb4o do not describe how dimensions are implemented, and which dimensions are part of the cube's key.

As for the missing information regarding the implementation of dimensions, the solution is quite simple. The implementation of dimension defines how the different levels of a dimension are arranged as table. The three possible implementations supported by qbgen are, `Degenerate`, `Star` and `Snowflake`. Within a degenerate dimension all dimension data is stored within the fact table. In a star dimension, all dimension data is stored within one dimension table. In a snowflake dimension each dimension level is stored in its own table. The type of implementation is the only additional information required for dimension generation. The type of implementation can be added to the definition of a dimension using the `qbgen:implementation` property. Listing 6 shows the three different implementations: It defines the date dimension as star dimension, the functional area dimension as degenerate dimension and the landscape dimension as snowflake dimension.

Listing 6: Example for qbgen:implementation Property

```
agri:Date_Dim a qb:DimensionProperty;
    qbgen:implementation qbgen:Star.

agri:FunctionalAreaDim a qb:DimensionProperty;
    qbgen:implementation qbgen:Degenerate.

agri:Landscape a a qb:DimensionProperty;
    qbgen:implementation qbgen:Snowflake.
```

Another type of information that cannot be declared using qb and qb4o is which of the referenced dimensions are part of a fact table's primary key. In the facts of the multidimensional model developed in the AgriProKnow project, a majority of the referenced dimensions is part of the primary key. Therefore, qbgen does not state which dimensions are in the primary key, it states the dimensions that are not part of it. To state such a dimension the property `qbgen:excludeFromKey` with the value `true` is added to the reference of a dimension level within the `qb:DataStructureDefinition` of a fact. Listing 7 shows the fact table calving, which references the dimension data, as the date dimension in the calving cube is only descriptive, although being a dimension, it is excluded from the key.

Listing 7: Example for qbgen:excludeFromKey Property

```
agri:Calving a qb:DataStructureDefinition;
    qb:component [
        qb4o:level agri:Date_;
```

```
        qb4o:cardinality qb4o:ManyToOne;
        qbgen:excludeFromKey true].
```

The automatic generation of a ROLAP schema requires a transformation rule from IRIs, which are used to denote qb and qb4o elements to valid names for relational schema elements. The sDWH employs as default rule for the generation of relational schema element names a simple transformation: The sDWH takes the substring of the IRI that comes after a specified prefix, `agri`[10] in the case of the AgriProKnow project. In some cases, however, the default transformation rule produces undesired results. For example, in the qb representation there exists a `Date_` dimension level with a `Date_Val` attribute. The `Date_Val` attribute should be represented as column with name "Date_" for usability's sake. Another reason for renaming is that a dimension may have a different meaning in one cube compared to another. For instance, if the calving cube references the animal dimension, it is not clear if the animal referenced is the calf or its mother (dam). Therefore, the animal dimension in the calving cube is renamed to `damanimal`.

In accordance to the examples used above, two types of renaming can be done, as illustrated in Listing 8. Renaming of an attribute within a dimension and renaming of a referenced dimension level within a cube. Both times a `qbgen:RenamingSet` is used. The renaming set hast two different properties. The `qbgen:rename` property states what should be renamed and the `qbgen:renameTo` property states its new name. A renaming set is then referenced by the environment the renaming takes place, which is either a dimension level (`qb4o:LevelProperty`) or the definition of a cube (`qb:DataStructureDefinition`).

Listing 8: Example for Renaming

```
agri:Date_ a qb4o:LevelProperty;
    qbgen:renaming [qbgen:rename agri:Date_Val;
            qbgen:renameTo "Date_"@en].


agri:Calving_CalfStatus a qb:DataStructureDefinition;
    qbgen:renaming [qbgen:rename agri:Animal;
            qbgen:renameTo "DamAnimal"@en].
```

The second reason for the creation of qbgen were requirements that arose during the development of the multidimensional sDWH model of the AgriProKnow project. In particular, the requirement for extended dimension attributes. During the creation of the multidimensional model of the sDWH, data were identified, which referenced multiple dimensions, but do not have the characteristics of a fact, as they may change over time. Furthermore, the attributes of these data were not measures, as they are only of a descriptive nature. Therefore `qbgen:ComplexAttributes` where designed. These are similar to cross dimensional attributes described by Golfarelli and Rizzi [12, p. 111] and pose a way to create data structures which are not part of a traditional multidimensional model. They can be linked to dimension levels using the `qbgen:linksDimension` property, the linked dimensions are always part of the primary key of a complex attribute. Moreover, it is possible to add additional attributes, either using the `qbgen:hasId` property to add attributes that are part of the primary key, or `qbgen:hasAttribute` to add non identifying attributes. Listing 9 shows the definition of the complex attribute reference curve. Reference curve includes the dimension calving no, the id attributes breed and day of lactation and the descriptive attribute reference val.

[10] `http://agriproknow.com/vocabulary/AgriPro#`

Listing 9: Example for the Definition of a Complex Attribute

```
agri:Reference_Curve a qbgen:ComplexAttribute;
    qbgen:linksDimension agri:CalvingNo;
    qbgen:hasID agri:Breed;
    qbgen:hasID agri:DayOfLactation;
    qbgen:hasAttribute agri:Reference_Val.

agri:Breed a qb:AttributeProperty;
    rdfs:range xsd:string.

agri:DayOfLactation a qb:AttributeProperty;
    rdfs:range xsd:integer.

agri:Reference_Val a qb:AttributeProperty;
    rdfs:range xsd:double.
```

As it is not only necessary to define the structure of complex attributes, but also to describe instance data, this is also part of the qbgen vocabulary. Instance data of a complex attribute is therefore of type `qbgen:ComplexAttributeInstance`. The property `qbgen:instanceOf` shows to which complex attribute the data belongs. To state the values of a complex attribute instance the properties defined in the complex attribute are used. Listing 10 shows data for the complex attribute reference curve. The shown complex attribute instance references the dimension level member `agrid:CalvingNo_1` and gives values for the attributes, breed, day of lactation and reference val.

Listing 10: Example for the Definition of Complex Attribute Data

```
agrid:Reference_Curve_1 a qbgen:ComplexAttributeInstance;
    qbgen:instanceOf agri:Reference_Curve;
    agri:CalvingNo agrid:CalvingNo_1;
    agri:Breed "Holstein";
    agri:DayOfLactation 1;
    agri:Reference_Val 25.
```

The last part of the qbgen vocabulary provides a fast-track facility for deletion of dimensions. Dimension are part of the primary keys of fact and complex attribute tables. If a referenced dimension is deleted which is part of the primary key a new primary key has to be defined for the table. To do this the key columns of all remaining dimensions, which are part of the primary key, have to be known. The fast-track facility for deletion of dimensions is similar to a drop force in relational database management systems as it bypasses integrity constraints. A new primary key is defined automatically for the remaining dimensions which are part of the key. This is problematic in cases where duplicates in the data arise due to the dropping of a key column. Thus, users should proceed with caution.

One option to modify the primary key of a fact or complex attribute table, would be to query for all key columns of the remaining dimensions. As dimensions are often part of different tables with different dimensions, the deletion of one dimension would result in more queries the more fact tables and complex attributes the dimension was part of. To accelerate this procedure the key columns of each fact table or complex attribute are added to the RDF schema using `qbgen:KeyColumnSet` on generation of the SQL schema. Each key column set has the property `qbgen:keyLevel`

stating the dimensions level the columns are part of and `qbgen:keyColName` stating all names of key columns. The key column set is referenced in the data structure definition by `qbgen:keys`. Listing 11 shows the key sets of the blood measurement cube. For each of the cube's dimensions, namely animal, date and farm site, a key column set is defined.

Listing 11: Example for the qbgen:KeyColSet

```
agri:BloodMeasurment a qb:DataStructureDefinition ;
        qbgen:keys [ qbgen:keyColName "NationalID";
                qbgen:keyLevel agri:Animal] ;
        qbgen:keys [ qbgen:keyColName "Date_";
                qbgen:keyLevel    agri:Date_] ;
        qbgen:keys [ qbgen:keyColName "FarmSiteId";
                qbgen:keyLevel    agri:FarmSite].
```

Qbgen, qb and qb4o are the basic vocabularies used in all web services regarding schema creation/modification, loading of instance data, and querying. The first web services to use these vocabularies are the web services for schema creation/modification which are described in the following section.

4.3 Loading Requests for Creation and Modification of the DWH Schema

The previous sections described the qbgen vocabulary and the loading procedure which is the same for schema and instance data. This section is solely focusing on the sDWH schema, which is created and modified using the loading procedure. As the multidimensional model of the AgriProKnow project is not predefined but developed during the project, it is important to provide capabilities to create the sDWH schema, and as the schema is evolving throughout the project it is equally important to provide modification capability.

To define the schema of the sDWH the RDF vocabularies qb, qb4o and qbgen are used. Based upon this RDF definition of the sDWH schema, and the loading process discussed in a previous section, the sDWH provides three RESTful web services to add loading requests for creation and modification of the sDWH schema. On execution of these loading requests the relational schema is generated or changes are made. These processes follow the principle of *convention over configuration*, as there is no explicit mapping between the RDF schema and the relational schema within the database. All mappings are implicit and encoded into the different loading requests. Therefore the loading requests for schema creation are discussed in the following section. They describe all implicit mappings between RDF and relational schema, as well as how the loading requests are issued.

4.3.1 Create Schema Elements Request

To create a schema in the sDWH the create schema elements request exists. In addition to the initial schema creation it is used to add further elements to an already existing sDWH schema. Using this loading request, it is possible to create dimensions, cubes and complex attributes. It is only possible to create these elements with the services. It purposely is not possible to create

any other elements like additional measures or additional dimension levels. These restrictions were chosen to prevent null values.

The create schema elements loading request follows the loading procedure discussed in Section 4.1. The dimensions, cubes and complex attributes to be created in the sDWH are defined in RDF format using the qb, qb4o and qbgen vocabularies. After the definition the data is sent as content of a HTTP POST request to the create schema elements web service at the URL `<server>/AgriProKnowDBService/rest/SchemaService/CreateSchemaElements`. If a create schema elements request is executed, the defined elements are generated in the following order: dimensions, fact tables of cubes and complex attributes.

Dimension Generation. At the beginning of dimension generation all elements of type `qb:DimensionProperty` and their `qbgen:implementation` property are queried from the loading request's named graph in the triple store. Each dimension is than individually processed. All levels of a dimension are queried from the triple store to create an appropriate representation of the dimension as Java objects. After a dimension is loaded, it is stored in a cache to accelerate the generation of fact tables and complex attributes referencing the dimension. During the loading of the dimension it is already checked for definition errors like wrong level hierarchies or missing key columns. If such a problem is found a corresponding exception is produced (see Table 4) and the execution of this loading request is ended.

The generation of the dimension continues according to its implementation. If the dimension is degenerate its generation ends here, as no table is created for it. We employ the term *degenerate dimension* to denote dimensions that are realised not in separate dimension tables but the levels and attributes of which are added to the fact table. Typically, degenerate dimensions consist of only one level, which is how literature employs the term [12, p. 252], [23, pp. 230 sqq.]. Note, however, that we use the term to specify a specific kind of implementation for lack of better term. All degenerate dimension levels are generated with the tables they are embedded in, which are either fact or complex attribute tables. In these tables all attributes of all dimension levels of a degenerate dimension are included as columns. If the dimension is of type star, one table is generated including all attributes of all dimension levels. The key attributes defined as `qb:AttributeProperty` and all describing attributes defined as `qb4o:LevelAttribute` within the dimension levels are generated as columns within the SQL script, whereas the name of the column is by convention defined as the IRI of the attribute without its prefix and the type is always defined by the value of its `rdfs:range` property. The mapping of the RDF data types used in attribute definitions and the column data types in the relational database is compliant to the W3C's recommendation in R2RML [7].

The table representing the star-type dimension has the name of its root dimension level. The name is by convention defined as the IRI without its prefix. Also the id attributes of the root level are the primary key of the dimension table.

If the dimension shall be implemented as snowflake-type its generation is more complex as each dimension level is implemented as own table. The table name is by convention the IRI of the `qb4o:LevelProperty` without its prefix, and the primary key is combined of the id attributes of the `qb4o:LevelProperty`.

Furthermore, cardinalities between those tables come into play. In addition to the many-to-one relation which is assumed in star and degenerate dimensions, the sDWH supports one-to-

one, many-to-many and one-to-many relations between dimension levels of snowflake dimensions. Many-to-many and one-to-many relations are only implemented in the schema creation and data loading part of the sDWH. This has been implemented to avoid restrictions in the development of the AgriProKnow multidimensional model. But as those relationships do not appear in the AgriProKnow model, they are not supported in the querying part of the sDWH (Section 5). To support these types of relationships further distribution strategies would be needed [21], which are not implemented in the current prototype.

The generation of a snowflake dimension can be seen as traversing of a tree. The dimension is traversed in post-order. Due to the nature of SQL this strategy is necessary, as all tables of parent levels of a child level have to be generated, for the table of the child level to be able to reference them. If the relation between a child and a parent level is of type many-to-one or one-to-one, the parent level is referenced through a foreign key referencing the table of the parent level. If the relationship is of type many-to-many or one-to-many group tables are needed. Therefore, two tables are created. First, a table containing the group id, using the naming convention "<child level>_<parent level>_group". This table is referenced by the child level's table. Furthermore, a table matching the group id to the parent level is created, the table is named using the convention "<child level>_<parent level>_" and is referencing the group id table and the table of the parent level.

After all star and snowflake dimensions are generated their scripts are added to the script of the loading request for later execution.

Fact Table Generation. The next step in execution of the create schema elements loading request is the generation of fact tables. As the dimensions are already generated, only the fact tables are missing to complete the defined cubes. For the generation, all elements of type `qb:DataStructureDefinition` are queried from the loading request's named graph in the triple store. Each fact is generated separately.

The first step, in the generation of a fact table is to generate all dimension references. Therefore, all referenced dimensions are loaded as Java objects, either from the dimension cache, or if not cached yet, from the triple store. If a referenced dimension is of type `qbgen:Degenerate`, all attributes of all its dimensions levels are added as columns to the SQL definition of the fact table. If the dimension is of type star or snowflake the key columns of the referenced dimension level are added to the fact table script and foreign key constraints referencing the tables are added. If the referenced dimension is of type star the reference is always to the root level of the dimension. If the referenced dimension is of type snowflake it is also possible to reference dimension levels other than the root level of the dimension.

Another important part in referencing star and snowflake dimensions is cardinality. The sDWH supports many-to-one, one-to-one, one-to-many, and many-to-many relations between fact and dimension. As mentioned one-to-many and many-to-many relations are only supported in schema generation and loading of data, not in querying. If the relation is one-to-one or one-to-many a reference to the dimension level is created as described previously. If the relation is of type one-to-one an additional unique constraint is generated over the columns referencing the dimension level, to ensure each dimension level entity is only referenced once. If the relation is of type one-to-many or many-to-many, the generation follows the same principles that are used during the generation of snowflake dimensions, whereas the fact tables can be seen as the child level, and the dimension level as parent level of the relation.

When all referenced dimensions are processed, measures and attributes are added as columns to the fact table script. Furthermore, a primary key constraint for the fact table is generated, that includes all key columns of the referenced dimension levels except of the dimensions that are marked with `qbgen:excludeFromKey true`. Furthmore, information about the key columns is added to the RDF schema in the triple store using the `qbdke:keys` property as described in section 4.2. If all dimensions are marked not to be included in the key, or if the fact table does not reference any dimensions, a `NoKeyColumnsException` is produced and the execution of the loading request ended. After all fact tables are generated their scripts are added to the script of the loading request for later execution.

Complex Attribute Generation. In the execution of a create schema elements loading request complex attributes are generated in the end. For the generationc all elements of type `qbgen:ComplexAttribute` are queried from the loading request's named graph in the triple store. Each complex attribute is generated separately. At the beginning all references to dimensions are generated. Therefore all referenced dimensions are loaded as Java objects, either from the dimension cache, or if not cached yet, from the triple store. The cardinality of the relation between complex attribute and dimension level is always many-to-one. Therefore all key columns of the referenced dimension level are added to the complex attribute table including corresponding foreign key constraints. After all referenced dimensions are processed the other attributes of the complex attribute are generated. In the end, a primary key constraint is created including all key columns of the referenced dimension levels and all attributes referenced by the property `qbgen:hasId`. If there are neither dimensions, nor attributes, that are part of the key a `NoKeyColumnsException` is produced and the execution of the loading request ended. After all complex attributes are generated their scripts are added to the script of the loading request for later execution.

The three steps mentioned are the main execution parts of a create schema elements loading requests. If no elements of a type are defined in the data of the loading request the part will be left out during execution of the loading request. After the scripts for all elements are created, they are executed in the database in the order they were generated. If the generation of all elements is successful the data of the loading request is added to the RDF schema of the triple store, so the RDF schema is equivalent to the SQL schema of the database. This concludes the create schema elements loading request, the next loading request provides functionality to add dimension attributes.

4.3.2 Add Dimension Attributes Request.

The add dimension attributes loading request provides the possibility to add descriptive attributes to a dimension level. Descriptive means that they are not part of the dimension level's key and therefore only of descriptive, not identifying nature, which is the reason why their addition is allowed. Descriptive dimension attributes are the only columns of the sDWH schema that allow null values, as a missing of these values does not effect the result of a query, it only results in the missing of additional descriptive information.

The add dimension attributes loading request follows the loading procedure discussed in Section 4.1. The dimensions, dimension attributes to be created in the sDWH are defined in RDF format using the qb, qb4o and qbgen vocabularies. Mainly the RDF types `qb4o:LevelProperty` and

qb4o:LevelAttribute, as well as the qb4o:hasAttribute property, are used to define the data. After the definition the data is sent as content of a HTTP POST request to the add dimension attributes web service at the URL <server>/AgriProKnowDBService/rest/-SchemaService/AddDimensionAttributes.

If an add dimension attributes loading request is executed each attribute is processed separately. The dimension, that an attribute is added to, is loaded from the dimension cache or the triple store. The next steps depends on the dimensions implementation. If the dimension is of type star or snowflake an ALTER TABLE statement is generated to add the attribute to the table representing the dimension/dimension level.

If the dimension is of type degenerate, generation is more complex, as all dimension data are embedded in the fact tables and complex attributes. Therefore all fact tables and complex attributes that include the dimension have to be queried from the RDF schema in the triple store. For each fact table or complex attribute that includes the dimension an ALTER TABLE statement for the fact table or complex attribute table is generated to add the attribute.

The last step in the execution of an add dimension attributes request is to clear all modified dimension from the dimension cache, as they need to be reloaded from the triple store. Furthermore, the definition of the new dimension attributes is added to the RDF schema in the triple store. The requests described until now deal with the creation of schema elements in the sDWH as opposed to the next request which deals with deletion of element.

4.3.3 Drop Schema Elements Request

The drop schema elements loading request provides the functionality to delete elements in the sDWH schema. As already in the create schema elements request these elements are dimensions, facts tables of cubes, and complex attributes. Only these elements can be deleted, it is not possible to explicitly delete only parts of these elements. Although an implicit deletion of parts may occur, e.g., if a dimension is deleted and it is referenced by a fact table or a complex attribute, the part referencing the dimension will be deleted from the element.

The drop schema elements loading request follows the loading procedure discussed in Section 4.1. The according web service has the URL <server>/AgriProKnowDBService/rest-/SchemaService/DropSchemaElements. All elements to be deleted in the sDWH have to be defined in RDF format using vocabularies qb and qbgen. The only properties necessary for the drop schema elements request are qb:DimensionProperty, qb:DataStructure and qbgen:ComplexAttribute. Using these RDF types, the elements to delete have to be listed in the data of the loading request. On execution of the loading request the stated elements are deleted in the following order: fact tables, complex attributes and dimensions.

Drop Fact Tables. For the drop of facts tables all elements of type qb:DataStructure-Definition are queried from the named graph of the loading request in the triple store. For each fact table to delete a DROP TABLE statement is generated, if the fact table has many-to-many or one-to-many relation to dimensions, DROP TABLE statements for all tables representing these relations are created. At last the fact table is added to the list of deleted objects for this

drop schema elements request. This list is used during dimension deletions to skip the altering of already deleted schema elements.

Drop Complex Attributes. The second elements, that are deleted are complex attributes. Therefore, all elements of type `qbgen:ComplexAttribute` are queried from the loading request data. As each complex attribute is represented as table in the database a `DROP TABLE` statement is create for each complex attribute stated. Moreover each deleted complex attribute is added to the list of deleted elements.

Drop Dimensions. The last part is the deletion of dimensions, which is more complex than the deletion of fact tables or complex attributes, as a dimension might be referenced by fact tables or complex attributes. It is not recommended to drop dimensions which are referenced by fact tables or complex attributes, as their deletion might lead to errors in the relational database or inconsistent data. The deletion of such dimensions, although not recommended, is possible. Because of the many risks of using this feature, future versions of the sDWH may introduce an exception.

To drop dimensions at first all elements of type `qb:DimensionProperty` are queried from the named graph of the loading request. Each dimension in the result is processed separately. The dimension is loaded from the dimension cache or the triple store.

Before the dimension itself is deleted all fact tables and complex attributes referencing the dimension are altered, with the exception of all elements that are part of the list of deleted elements of this loading request. To change the facts and complex attributes, `ALTER TABLE` statements are generated which drop the columns in the fact or complex attribute tables referencing the dimension table; or if it is a degenerate dimension, drop all columns of the dimension in the table the degenerate dimension is part of. If the columns dropped were part of the primary key, a new primary key has to be generated. Therefore, all primary key information provided by `qbgen:keys` is queried from the database and used to create a new primary key constraint. If the dimension to delete is a degenerate dimension, this ends the dimension deletion. If the dimension to drop is of type star or snowflake and the element to alter is a fact table that involves a many-to-many or one-to-many relation between the fact and the dimension to delete, `DROP TABLE` statements for all tables representing this relation are created as well.

If the dimension to delete is a star or snowflake dimension the dimension tables representing the dimension have to be dropped as well. If the dimension is a of type star, only one `DROP TABLE` statement is necessary. If the dimension is of type snowflake generation of `DROP TABLE` statements starts at the root level of the dimension, continuing with its parent levels. If a child and a parent dimension level are in a many-to-many or one-to-many relation, `DROP TABLE` statements are generated for all tables representing the relationship.

After generation of all statements, all deleted, fact tables, complex attributes and dimension are also deleted from the RDF schema stored in the triple store. This is important as the RDF schema should represent the schema in the database.

4.4 Loading Requests for Instance Data

The previous section described loading requests for the creation and modification of the sDWH schema. These are a prerequisite for loading of instance data, as data are specified using the defined schema and as database tables are created on execution of the create schema elements loading request, into which the instance data is loaded. Because of these reasons schema creation is important and necessary, but it is only a means to provide structure for the data to be analysed.

To load data into the sDWH four types of loading requests are provided. The insert/replace and the insert/replace for snowflake data requests add new data to the sDWH or modify existing data. The requests delete and delete relation data remove data from the sDWH. A single loading request can only change data in either, one dimension, fact table or complex attribute. To specify which element shall be changed the elements name is added to the web service URL. Other than the RDF schema data, the instance data in RDF format are not stored after they were loaded into the sDWH, as they are no longer needed. All loading requests follow the loading procedure defined in Section 4.1 and use the qb, qb4o and qbgen vocabularies. A more detailed description of the requests can be seen in the following sections.

4.4.1 Insert/Replace Request

The first loading request for loading of data is the insert/teplace request. This request combines the SQL commands `INSERT` and `UPDATE`. All data for this request are defined based on the RDF schema of the sDWH which uses qb, qb4o and qbgen. On execution of the request, data are updated if existing data records can be matched to new data records. If no existing data records are matched the data are inserted as new. This is done using Oracle's `MERGE` statement [18]. Within one loading request data for `INSERT` and `UPDATE` can be mixed. The upload of an insert/replace loading request is done over the web service with the URL `<server>/AgriProKnowDBService/rest/InstanceService/InsertOrReplaceData/`. At the end of the URL the element which shall be modified must be stated. This element is either a `qb:DimensionProperty`, `qb:DataStructureDefinition`, or `qbgen:ComplexAttribute`. Only the name of the element is added, which is by convention the IRI of the element without its prefix. Depending on the type of the element the execution of the loading request differs, although on the outside the procedure is the same.

Insert/Replace Dimension Data. Before fact table and complex attributes, data of tables referencing snowflake or star dimensions is loaded into the sDWH, the data of said star and snowflake dimension have to be loaded. Degenerate dimension data are added with the data of according fact tables and complex attributes. Depending on the implementation of the dimension the execution of the loading requests differs.

The loading of star dimension data is rather simple, as all data are stored in one table and therefore can be added using one `INSERT` statement. For star dimension data to be loaded, all data of all dimension levels have to be present in the loading request's data. The loading of data is based on two statements. First a SPARQL statement is generated which selects all data for insertion. Each entry returned by the SPARQL `SELECT` is one entry in the sDWHs database and each result column of the `SELECT` is one column of the table representing the dimension in the sDWH. According to this a `MERGE` statement is generated to insert and replace the new data.

On execution of the SPARQL `SELECT` each row of the result is added as `MERGE` statement for batch processing. Every 1000 rows or after all data is processed the batch is executed and the data added to the sDWHs database. Listing 12 shows instance data for the animal dimension levels animal and main by providing a member of the animal level with attributes animal name and national id as well as referencing a member of its parent level main breed.

Listing 12: Example for Data of Star Dimension AnimalDim

```
agrid:Kuh_1  a                qb4o:LevelMember ;
        agri:AnimalName       "NAGERL" ;
        agri:NationalID       "AT00000000123" ;
        agri:hasMainBreed     agrid:Holstein-Schwarzbunt ;
        qb4o:memberOf         agri:Animal .

agrid:Holstein-Schwarzbunt
        a                  qb4o:LevelMember ;
        agri:Breed      "Holstein-Schwarzbunt" ;
        qb4o:memberOf   agri:MainBreed .
```

If the dimension is a snowflake dimension the loading of data is more complex, as dimension data are scattered among multiple tables and cardinalities between those tables have to be considered. Therefore, loading of data is started with the dimension levels that do not reference any other levels. Loading of a snowflake dimension level follows the same principal as loading of a star dimension. All relevant columns, which are all the attributes of the level, are selected from the triple store and added to according `MERGE` statements. If the first levels have been loaded, it is checked which of the remaining levels can be loaded next. If a child level is referencing only parent levels that have already been processed, loading is possible for the child. Other than in star dimensions where RDF data for all levels of the dimension are needed, in snowflake dimensions it is possible to only add data for some dimension levels. Therefore the RDF data have to be complete for the levels to be added and include the key attributes (`qb4o:hasId`) of the referenced parent level entities .

The only additional step in the loading of a snowflake dimension occurs if a child and a parent level have a one-to-many or many-to-many relationship. This relation is represented by two tables, the group-id table and the group table. The child level is referencing the group-id table. The group table is referencing the group-id table and the parent level. At the beginning of loading many-to-many or one-to-many relation data, it is checked if the child-level entities which have to be loaded, are already referencing a group-id. For each new entity a new and unique group-id is created. Then the group-id and the referenced parent levels for each child-level entity are added to the group table. During the insert/replace request only new entities are added to an existing group. Parent-level entities which are already part of the group are untouched. To remove parent-level entities from a one-to-many or many-to-many relation between dimension levels the delete relation data request has to be used (see Section 4.4.3).

The one-to-many and many-to-many relation data is inserted into the sDWH before processing the data of the child level of the relationship. If the relation data and the child level data are loaded into the database the loading of the snowflake dimension continues with the next levels. If the root level of a dimension is loaded the loading of the dimension is complete and the loading request ends.

Insert/Replace Fact Table Data. When the referenced star and snowflake dimension data were added to the sDWH it is possible to add fact table data, including data for degenerate dimensions referenced by the fact table. The loading request for the fact table data have to include a `qb:DataSet` referencing the `qb:DataStructureDefinition` of the fact table and `qb:Observations` representing the data to add, which reference the data set. The observations have to include references to dimension level data as well as data for measures and attributes defined in the schema for this fact table. If one attribute is missing, the observation is not valid and will not be processed. Furthermore, data for the referenced dimension levels in form of `qb4o:LevelMembers` have to be included. If the referenced level is part of a snowflake or star dimension, only values for its key attributes (`qb4o:hasId`) have to be stated within the level member. If the dimension is degenerate, data of all dimension levels and attributes have to be included, as the dimension data are stored in the fact table. An example can be seen in Listing 13, where the entities for dimension levels date, animal and farm site only contain the key attributes, as they are star dimensions.

Listing 13: Example Data of an Insert/Replace Request for the Fact Calving

```
agrid:DataSet_Calving_1
        a               qb:DataSet;
        qb:structure  agri:Calving.

agrid:Calving_1  a        qb:Observation ;
        agri:Calf         agrid:Calf_1;
        agri:CalfStatus   "OK";
        agri:CalfWeight   12.5;
        agri:CalvingEase  2;
        agri:CalvingNo    agrid:CalvingNo_2;
        agri:Animal       agrid:Kuh_1;
        agri:Date_        agrid:2016-02-03;
        agri:FarmSite     agrid:FarmSite_1;
        qb:dataSet        agrid:DataSet_Calving_1.

agrid:Kuh_1  a           qb4o:LevelMember;
        agri:NationalID  "AT000000000123";
        qb4o:memberOf    agri:Animal.

agrid:CalvingNo_2  a    qb4o:LevelMember;
        agri:CalvingNo_Val  2;
        qb4o:memberOf   agri:CalvingNo.

agrid:FarmSite_1  a      qb4o:LevelMember;
        agri:FarmSiteId  "12345";
        qb4o:memberOf    agri:FarmSite.

agrid:2016-02-03
        a                qb4o:LevelMember;
        agri:Date_Val  "2016-02-03";
        qb4o:memberOf  agri:Date_.

agrid:Calf_1  a        qb4o:LevelMember;
```

```
        agri:Calf_Val  1;
        qb4o:memberOf  agri:Calf.
```

The execution of an insert/replace loading request with fact table data works similar to the execution of a request with data of a star dimension as both only consist of one table. It differs if the fact table has one-to-many or many-to-many relations with one or more star or snowflake dimensions. These are processed before other data of the fact table. Therefore, all one-to-many and many-to-many relations are processed sequentially, starting with relations which are part of the key. At first, each observation is checked if it corresponds to an entry existing in the database. This is only possible if no many-to-many relations are part of the key. Because the AgriProKnow project does not require this feature, the sDWH only provides rudimentary support for this feature. If a many-to-many or more then one one-to-many relations are part of the key an update of the fact table is not possible and all data sent with an insert/replace loading request are inserted as new data. If this restriction does not occur update is possible and new data are added to existing one-to-many, many-to-many relations. Either way, the one-to-many and many-to-many relations are treated the same as relations between snowflake dimension levels. During processing of the one-to-many/many-to-many relations the group-ids of the relations which are generated or queried are saved for each entity. This data is used in the second step.

In the second step, two statements are generated. A SPARQL `SELECT` statement that brings the RDF data into table form, with all necessary columns and a SQL `MERGE` statements that loads each observation as entry in the fact table of the sDWH's database. Both are generated from the schema data in the triple store. Only observations that include all measures, attributes and dimensions are considered, all others are not considered by the SPARQL `SELECT`. On execution of the `SELECT` each row of the result is added as `MERGE` statement for batch processing. If there are one-to-many or many-to-many relations the group-ids generated in the first step are used in addition to the SPARQL `SELECT` result. Every 1000 rows or after all data are processed the batch of `MERGE` statements is executed and the data added to the sDWHs database. If all selected observations have been processed the execution of the loading request ends.

Insert/Replace Complex Attribute Data. Complex attribute data have to be defined using the RDF type `qbgen:ComplexAttributeInstance`, in addition data for the referenced dimension level instances have to be given (see Listing 14). For star and snowflake dimensions only the key attributes have to be stated. For degenerate dimensions all attributes of all dimension levels must be included. The loading of complex attribute data is rather simple, as all data are stored in one table and there are only relations to dimension levels with the cardinality many-to-one. Therefore, it is possible to load the complex attribute data with only two statements. A SPARQL `SELECT`, of which each result is one entry in the complex attribute table and a SQL `MERGE` statement to add the data to the database. On execution of the SPARQL `SELECT` each row of the result is added to a `MERGE` statement for batch processing. Every 1000 rows or after all data are processed the batch is executed and the data added to the sDWHs database. If all rows of the `SELECT` are processed the loading request is ended.

Listing 14: Example Data of an Insert/Replace Request of the Complex Attribute Enterprise

```
agrid:Enterprise_1  a     qbgen:ComplexAttributeInstance;
        qbgen:instanceOf agri:Enterprise;
        agri:End          "2051-04-25";
        agri:EnterpriseId  "77777" ;
```

```
        agri:FarmSite      agrid:FarmSite_1 ;
        agri:Start         "2001-04-25" .

agrid:FarmSite_1  a      qb4o:LevelMember ;
        agri:FarmSiteId  "12345" ;
        qb4o:memberOf    agri:FarmSite .
```

The insert/replace loading request adds data or updates existing one, but if wrong data was loaded that cannot be corrected through modification, it needs to be deleted. This can be done using the delete loading request described in the next section.

4.4.2 Delete Request

The delete loading request is used to remove data from the sDWH, to do so it uses the SQL `DELETE` statement. All data to delete have to be defined using qb, qb4o and qbgen. The defined data only need to include identifying attributes. A delete loading request can be issued using the web service URL `<server>/AgriProKnowDBService/rest/InstanceService/DeleteData/` As only data of one element can be deleted in a single loading request the name of the element must be added at the end of the URL. This element is either of type `qb:DimensionProperty`, `qb:DataStructureDefinition` or `qbgen:ComplexAttribute`. Only the name of the element must be added, which is by convention the IRI of the element without its prefix. The delete loading request is destructive to the data in the sDWH and once executed its effects cannot be removed. Depending on the type of the element, the execution of the loading request differs, although on the outside the procedure is the same. If the data deleted is referenced by another element, e.g., dimension data referenced in facts, the entities referencing the deleted data are deleted as well, as no null values are allowed.

Delete Dimension Data. The deletion of dimension data depends on the `qbgen:ImplementationType` of a dimension. Data of degenerate dimensions are deleted with the data of the fact table or complex attribute table the dimension is part of. The deletion of star dimension data is rather simple. To delete data of a star dimension only the entities for the root levels of the star dimension and the key attributes of this level must be provided (see Listing 15). The ids of the entities to delete are queried from the loading request data using a SPARQL `SELECT` statement. The ids are then used to fill SQL `DELETE` statements. SPARQL `SELECT` and SQL `DELETE` statements are generated based on the RDF schema in the triple store. As all foreign keys in the sDWH's SQL schema are cascading, all references on the deleted dimension are deleted automatically. This may result in deletion of fact or complex attribute entries.

Listing 15: Example Data for a Delete Request of Star Dimension Animal

```
agrid:Kuh_1  a qb4o:LevelMember;
    qb4o:memberOf agri:Animal;
    agri:NationalID "AT00000001111".

agrid:Kuh_2  a qb4o:LevelMember;
    qb4o:memberOf agri:Animal;
    agri:NationalID "AT00000001112".
```

If there are any one-to-many or many-to-many relations between a fact table and a dimension additional steps have to be taken. As the fact table references the group-id table and the group table references the group-id table and the dimension table, the deletion of dimension data may lead to an empty group, where a group-id is referenced by the fact, but is not referenced in the group table. These empty groups are delete from the group-id table, which results in deletion of fact table entries. If the relations are cleared of empty groups, the deletion of star dimension data is finished.

As snowflake dimensions are scattered over multiple tables, and cardinalities have to be considered, the deletion of snowflake dimension data is more complex. The data to be deleted include all levels of the dimension. As long as all key attributes (`qb4o:hasId`) in the data of a level are provided, the data is deleted. The deletion of each level of the snowflake dimension follows the same recursive procedure. At first the deletion in all its parent levels is triggered. When all parent level data are deleted, all one-to-many and many-to-many relations to parent levels are checked for empty groups, which is the same as in the one-to-many, many-to-many relations of a fact table and a dimension. After all relations to parent levels are processed, the data of the snowflake level itself are deleted. Therefore a SPARQL `SELECT` queries all data to delete, and from its result `DELETE` statements are generated which are added to a batch. After 1000 statements or after all data of this level is processed the statements are executed. After all data of this level is processed once more the one-to-many and many-to-many relations are checked. This time they are checked for group-ids which are no longer referenced in the current dimension level. These group-ids are deleted, which results in a deletion of all referencing entries in the group table. The last step of data deletion in a snowflake level is to check all fact tables which are in a one-to-many or many-to-many relation with the dimension level. This step is equal to the last step of star dimension data deletion. After all levels finish this process the execution of the loading request ends.

Delete Fact Table Data. For the deletion of fact tables `qb:Observation`s have to be stated which include data for all referenced dimension levels that are part of the key. For each entity of a dimension level only the key attributes have to be provided. If a fact table has one-to-many or many-to-many relations to a key dimensions the deletion starts with querying the according group-ids of the observations to delete from the sDWH's database. Those are needed to identify the entries in the fact table. To get all other identifying attributes a SPARQL `SELECT` statement is generated. After execution of the `SELECT`, the results are combined with the group-ids selected in the first step to create a `DELETE` statement for each observation, which is added to a batch. The batch is executed every 1000 statements or if all observations are processed. If all data was deleted from the fact table all many-to-many or one-to-many relations to dimensions are checked, as there might be group-ids which are no longer referenced in the fact table. If this is the case they are deleted which also removes the referencing entries in the group tables. After all relations are checked the execution of the loading request ends.

Delete Complex Attribute Data. The deletion of complex attribute data is rather simple as all data are stored within one table. To specify the data to delete `qbgen:ComplexAttribute-Instance`s are used, which have to include all attributes part of the key (see Listing 16), which are, data for all referenced dimensions and data for all attributes connected with `qbgen:hasId`. On execution of the loading request a SPARQL `SELECT`, of which each result is one entry in the complex attribute table, and a SQL `DELETE` statement to remove the data in the database, are generated. On execution of the SPARQL `SELECT` each row of the result is added as `DELETE`

statement for batch processing. Every 1000 rows or after all data are processed the batch is executed and the data removed from the sDWHs database. If all rows of the `SELECT` are processed the loading request is ended.

Listing 16: Example Data for a Delete Request of Complex Attribute Reference Curve

```
agrid:Reference_Curve_1  a qbgen:ComplexAttributeInstance;
    qbgen:instanceOf agri:Reference_Curve;
    agri:CalvingNo agrid:CalvingNo_1 ;
    agri:Breed "Hohlstein";
    agri:DayOfLactation 2.

agrid:CalvingNo_1 a qb4o:LevelMember ;
    agri:CalvingNo_Val 1;
    qb4o:memberOf agri:CalvingNo .
```

The loading requests described until now are applicable for all elements of the sDWH schema. The following requests are solely for snowflake dimensions.

4.4.3 Additional Loading Requests for Snowflake Dimensions

The two loading requests described in this section are by-products of the insert/replace and delete loading requests dealing with snowflake dimension data. They are only applicable for dimension of type `qbgen:Snowflake`. As they are not used within the AgriProKnow project, they are only described briefly.

Insert/Replace Snowflake Dimension Level Data Loading Request. This loading request is used to insert data into one level of a snowflake dimension. It works the same way as the loading of snowflake level data during an insert/replace loading request, only it is restricted to one level of the dimension. Therefore, see Section 4.4.1 for further details of the process.

For this request the data of the level to be inserted have to be defined using qb, qb4o and qbgen. In addition entities for the referenced parent dimension levels with their key attributes have to be defined. Each insert or replace snowflake dimension level data request can only load data into a single level of a dimension. The name of the level has to be added at the end of the web service URL, which is `<server>/AgriProKnowDBService/rest/InstanceService-/InsertOrReplaceSnowflakeDimensionLevelData/`

Delete Relation Data Loading Request. This loading request is only applicable for snowflake dimensions which include one-to-many or many-to-many relations between dimension levels. It deletes data in these relations, without explicitly deleting data of the dimension levels themselves. To specify relation data to delete, the entities of the child and parent levels have to be included in the loading request data. Both child and parent need their key attributes and the child needs to reference the parent level. The name of the dimension has to be added to the end of the web service URL to issue the loading request. The URL of the web service is `<server>/AgriProKnow-DBService/rest/InstanceService/DeleteRelationData/`

The process of data deletion is similar to a delete request for snowflake data. The levels are processed recursively. When a level is processed it firsts starts the deletion of relation data in all its parent levels. Afterwards the relation data between the current level and its parents, stated in the loading request, are delete. Closing the deletion within a level are the deletion of group-ids which are either not referenced in the group table or the child level table. How this process works is described in more detail in Section 4.4.2. In this process entities of the dimension level may be deleted, therefore, fact and complex attribute table data may be deleted as well and if there are one-to-many or many-to-many relations between facts and the dimension level, the according relation data has to be checked, a detailed description of this process can also be found in Section 4.4.2. After all levels of the snowflake dimension are processed the loading request ends.

4.5 Conclusion on Schema Definition and Data Loading

This section described the creation of the sDWH schema and how instance data are loaded into the sDWH. The conventions we chose to generate a relational schema out of a RDF schema are only one possible alternative. The ways of schema creation might be changed in the future to better represent the RDF schema in the relations or to introduce an explicit mapping instead of conventions.

One alternative for a better representation could be to introduce surrogate keys which represent the IRIs of the RDF objects, either by saving the IRI in them or by combining the id attributes into one identifying value. This surrogate key could have the same name as the object it identifies, which might make things clearer, as now the objects are only represented by their key columns which have a different name than the object itself.

The schema creation is a step in the development of the AgriProKnow sDWH which will be used rarely, if the schema has reached a certain degree of completeness, however, loading of data will happen throughout the project, as more data is produced daily in the farms which needs to be added so it can be used in analysis, which is the other main part of this thesis and is described next.

5 Data Analysis

This section describes the other main part of this thesis, the analysis of the data in the sDWH. The creation of the sDWH schema and the loading of instance data are steps before analysis of the data, but analysis is the important part as it uses the data already present in a new way. The main goal regarding data analysis in the AgriProKnow project is that the creation of analytical queries should be rather simple. It is the goal to provide querying capabilities to non-experts in database technology, therefore, it is important to provide means of simplifying queries. An additional but not mandatory goal is that the results of queries can be exported in RDF format using the qb and qb4o vocabularies. This would provide the queried data in a common format, so it can be used in other systems, moreover, the query results could be enriched with semantic information.

To reach these goals certain steps are taken. The first of these steps, is to restructure the schema of the sDWH to ease analysis of the data. The restructuring results in the analysis view. During the creation of the analysis view cubes of the sDWH are combined to subject-oriented views, namely the analysis tables. Furthermore, derived information is added to the new analysis tables.

The analysis view already eases querying, but the main factor in simplifying querying is to simplify the way a query is created. To find such a way we analysed example queries defined by domain experts who are part of the AgriProKnow project. The analysis revealed certain patterns, which were recurring in these examples. Based on this analysis we created semOLAP patterns, which ease the creation of queries. SemOLAP patterns are defined by experts in database technology, but they can be used by non-experts in database technology. A detailed description of semOLAP patterns will be given in a later section.

In the following section the first step of analysis the analysis view will be described. This includes how the analysis view can be configured and how it is generated.

5.1 The Analysis View

The analysis view adds subject-oriented views to the sDWH schema through combination and completion. It is optimized for analysis, as fact tables of the sDWH schema are combined, enriched with calculated information and missing data is partly completed. The analysis view was developed for the multidimensional model of the AgriProKnow project and relies on some data specific to this model, e.g., calving data. To create the analysis view it has to be configured using RESTful web services. After configuration its generation is also triggered by a RESTful web service, but the generation itself happens within a PL/SQL procedure and results in the creation of materialized views within the sDWH's database. In the next section a detailed description of the configuration possibilities will be given.

5.1.1 Configuration of the Analysis View

The analysis view is flexible and has to be configured using RESTful web services. As the configuration data is rather simple it is provided as comma separated values to the web services. The services can be used to configure: analysis tables, analysis groups, continuation and virtual analysis columns. There are two web services for configuration `<server>/AgriProKnow-`

DBService/rest/AnalysisViewConfigService/InsertReplaceConfig/ and <server>/AgriProKnowDBService/rest/AnalysisViewConfigService/DeleteConfig/. Both of these services are called using HTTP POST requests and data in CSV format using a semicolon as separator. The insert replace config web service either inserts new entries into the configuration or overwrites existing ones. The delete config service deletes entries from the configuration. All deletes are cascading, therefore, parts of the configuration which reference the deleted data are deleted as well. At the end of both web service URLs the type of data to add or delete has to be stated. Which is either AnalysisTable, AnalysisGroup, Continuation or VirtualAnalysisColumn.

The main part of the analysis view are the analysis tables. The analysis tables are the outcome of analysis view generation. Analysis tables are the center node for all configurable parts of the analysis view and therefore have to be configured first. Analysis tables can be added using the web service described above with and the addition of AnalysisTable at the end of the URL. Each entry of the analysis table configuration is one line in the submitted CSV data and has two columns. The name the analysis table is given in the first column, and either the value 1 or 0 in the second column. If the second column is 1, calving information is added to the analysis table during generation. The calving data consists of the attribute DayOfLactation and the dimension CalvingNo. The DayOfLactation column calculates how long an animal (a cow) has been lactating for. Lactation begins when a cow calved. The CalvingNo dimension has one column and states how often a cow calved. Both are derived from data of the analysis table Calving, which is only possible if the cubes of the analysis group associated to the analysis table reference the Animal and the Date dimension.

The analysis groups are the second part of the analysis view configuration. Each analysis table consists of at least one cube of the sDWH schema. All cubes which are part of an analysis table are considered an analysis group. They are configured adding AnalysisGroup at the end of the configuration web service's URL. The CSV data for configuration have two columns. The first column, is the name of the analysis table that identifies the group. The second column, is the name of the cube which is part of the analysis group. All entries with the same analysis table are combined into this analysis table on generation. Therefore it is required that all cubes of the sDWH schema within one analysis group reference the same dimensions and do not have any measures with the same name. Analysis tables and analysis groups are mandatory in the analysis view configuration.

An optional feature of the analysis view is continuation. Continuation can be defined for each cube of the sDWH schema. It only takes effect if a cube is part of an analysis group. Continuation eases aggregation in analysis. All of the cubes in the AgriProKnow multidimensional model include an animal, a date and sometimes an hour dimension, which means each entry for a measure in this cubes is associated to an animal and a certain period of time. If aggregation of measures is done over a bigger period of time, e.g., a month the result of the aggregation operation might be faulty if entries for some smaller time periods, e.g., days, are missing within the bigger time period. Therefore, continuation is used as form of completion for the data. The gaps in the data are closed by continuing with an older value until a new value exists or a certain amount of time passes. For instance, if continuation happens for 4 days, day 1 has the value 5, day 8 has value 3. The days 2-5 get the value 5 as it is continued from day 1, 4 days long. The values are not endlessly continued as they might not be valid any more after a certain amount of time. If continuation is added to a cube that is part of an analysis table, the analysis table gets an additional column with the name of the cube and the suffix "_CT". This columns states how old the continued values of each

entry are. If this value is `0`, the value is original and was not created through continuation. To configure continuation for a cube of the sDWH schema `Continuation` is added at the end of the configuration web service's URL. The configuration data have two columns. The first column, states the name of the cube which should have continuation. The second column, is an integer value and defines how long a value is continued. Which unit of time this value represents depends on the cube.

Continuation only happens with entries dated in the past, until the current date. It does not project values into the future. Continuation does not only stop if the maximum time defined is reached. Furthermore, if an animal leaves a farm site continuation is ended for the animal in this farm site. As a change of farm site has a big influence on an animal, the past values are not usable on a new farm site. If an animal leaves a farm site an exit event for the animal is recorded in the `Occurred` cube.

The last part of the analysis view are virtual analysis columns, which are only optional parts of an analysis table. A virtual analysis column is bound to an analysis table. As of the semantic differentiation, the same measurements from different sources are stored in separate columns, which helps to determine the origin of data, but for some queries this origin is not as important, therefore virtual analysis columns help to reverse the differentiation. They combine the values of a measurement which is stored in multiple measures (as it came from different sources) into a new measure. Therefore measures which are to include into the virtual analysis column are specified in an order. The order is crucial for the value of the virtual analysis column. The virtual analysis column checks each associated measures within a row in the specified order. If the measure checked is null, it moves on to the next, but if the measure is not null its value is adopted as the value of the virtual analysis column. If one of the measures included in the virtual analysis column is located within a cube with continuation, a column to state how old the continued values are is added. Therefore if the value of the virtual analysis column is from continuation, this column states how old the value is. The name of this column is the name of the virtual analysis column with the suffix "_CT".

The virtual analysis column's name has to be unique within the analysis table. Virtual analysis columns are configured using the configuration web service with the addition of `VirtualAnalysisColumn` at the end of the URL. The configuration data have five columns: the analysis table it is added to, the name of the virtual analysis column, the order within the virtual analysis column, the name of the cube in the sDWH schema the included measure is part of, and the name of the measure to include.

For the configuration of the analysis view or changes of it to take effect, the analysis view has to be generated, which is described in the next section.

5.1.2 Creation of the Analysis View

The analysis view is created using a RESTful web service. After analysis table and analysis groups are configured the web service can be called with a HTTP GET request to the URL `<server>/AgriProKnowDBService/rest/AnalysisViewConfigService/GenerateViewSchema/`

The web services starts a stored procedure within the database of the sDWH. This stored procedure reads the analysis view configuration and at first deletes all the existing analysis tables. It then generates a script for each defined analysis table by combining cubes, adding continuation, virtual analysis columns and columns with calving information, if defined. Each analysis table is created as materialized view in the database. They only analysis table that is not generated by the analysis table generator is the analysis table `Calving`, because of its peculiar structure. The `Calving` table is fundamental to the other analysis table that include calving data and must exist prior to generation of other analysis tables. The calving table is very specific to the AgriProKnow project and rather stable, therefore, the script for the analysis table calving is created manually and the `Calving` table already has to exist in the sDWH's database when the creation of the analysis view is started. Furthermore, it is not deleted at the beginning of analysis view creation.

As the analysis tables are materialized views, the data in the analysis tables has to be reloaded regularly. For instance, if new data is loaded into the sDWH, for it to take effect when querying, a refresh of the analysis tables is necessary. This is done with a HTTP GET request to the web service with the URL `<server>/AgriProKnowDBService/rest/AnalysisView-ConfigService/GenerateView/`

5.1.3 An RDF Vocabulary for Analysis View Configuration

Currently CSV is used as format for the configuration of the analysis view. As the structure of the configuration data is very simple and as only the result of analysis view generation the subject-oriented views in the sDWH are used, there is no need for a more complex description of the analysis view, at this point, however, in the future it could be interesting to describe the analysis view in RDF. To do so would be beneficiary to the RDF schema of the sDWH, as all structures within the sDWH's database would be described in RDF format. Furthermore, the RDF definition of the analysis view could be used for integrity checks, that could be done during query creation, e.g., to check if a measure could be used within an analysis table. In this section we propose a first idea of a vocabulary to describe the analysis view. The vocabulary is based on the defined structures within the sDWH and references its RDF schema.

The first part of this new vocabulary is the definition of analysis table, an example can be seen in Listing 17. In this example the analysis table `BodyCondition` is defined. Using the property `consistsOf` all cubes are referenced which are part of the analysis group of body condition. Each entry has three properties. The property `baseCube` to state which cube should be included; the property `coninuationSpan` to state over which time span a value should be continued; the property `continuationColumn` to state in which column the value is stored how long the measures of the cube have been continued. This column has to be defined as `qb:Attribute` with range integer and should end with the suffix "_CT" for continuation, whereas the base cube property is mandatory, all other properties are only optional. In addition to the cubes, the analysis table has references to attribute `agri:DayOfLactation` and dimension `agri:CalvingNo`, if calving data are added.

Listing 17: Example of a possible RDF Definition for Analysis Table Body Condition

```
agri:Bodycondition a :AnalysisTable;
    :consistsOf [
        :baseCube agri:BodyCondition_BCS;
        :continuationSpan 20;
```

```
        :continuationColumn agri:BodyCondition_BCS_CT];
    :consistsOf [
        :baseCube agri:BodyCondition_BFT;
        :continuationSpan 20;
        :continuationColumn agri:BodyCondition_BFT_CT];
    :consistsOf [
        :baseCube agri:BodyCondition_Weight;
        :continuationSpan 20;
        :continuationColumn agri:BodyCondition_Weight_CT];
    qb:component [
        qb4o:level agri:CalvingNo;
        qb4o:cardinality qb4o:ManyToOne;
        qbgen:excludeFromKey true];
    qb:attribute agri:DayOfLactation.

agri:BodyCondition_BCS_CT a qb:AttributeProperty;
    rdfs:range xsd:integer.

agri:BodyCondition_BFT_CT a qb:AttributeProperty;
    rdfs:range xsd:integer.

agri:BodyCondition_Weight_CT a qb:AttributeProperty;
    rdfs:range xsd:integer.
```

In addition to analysis table, analysis group, and continuation described above, the vocabulary should also be used to define virtual analysis columns as seen in Listing 18. The virtual analysis column `Milkyield` show in Listing 18 includes three measures. Each of them is referenced with the `includesMeasure` property. In each reference the `analyseMeasure` property references the measure and the `order` property defines the order of the measures. Using the `consistsOf` property it can be referenced by an analysis table.

Listing 18: Example of a possible RDF Definition for the Virtual Analysis Column Milkyield

```
agri:Milkyield a :VirtualAnalysisColumn;
    :includesMeasure [
        :analyseMeasure agri:Milkyield_Parlour;
        :order 1];
    :includesMeasure [
        :analyseMeasure agri:Milkyield_DHIA;
        :order 2];
    :includesMeasure [
        :analyseMeasure agri:Milkyield_Dairy;
        :order 3].
```

This section concludes the description of the analysis view. The analysis view is only a means to ease querying, which is discussed in the next section.

5.2 Introducing semOLAP Patterns

One of the goals of the AgriProKnow project is to provide a possibility for simpler analytical queries. By structuring, preprocessing the data, and loading it into the sDWH, the analysis of the data is already eased. Furthermore, the generated subject-oriented views help to ease querying, as they combine and enrich the data. However the main complexity in querying is the query language. As data in the sDWH are stored within a relational database, the query language to use is SQL. The problem is, that SQL has to be learned before it can be used, which is a more time consuming task as not only the language itself needs to be understood but also the concept of relational databases. Moreover, to use SQL within a relational database extensive knowledge of the database's schema is necessary. These are all preconditions which are only met by experts in database technology, but not the average users targeted by the AgriProKnow sDWH. Therefore a simpler way of using SQL had to be found.

To find such an alternative we started by looking at queries that would be executed in the sDWH. We gave domain experts within the AgriProKnow project the task to create example queries in textual form. The result of this task can be seen below.

Query 1. Query for a specific day the number of all cows of a farmsite which match the description: lactation started within the last 30 days; milk has a high fat content according to DHIA (Dairy Herd Information Association); the value for fat content has not been a continuation for more than 30 days. Compare this number to the number of all cows of the farmsite.

Query 2. Calculate the Delta-BCS (Body Condition Score) value for all cows within a certain time period. The Delta-BCS calculates by subtracting the BCS value from day 30 of lactation, from the BCS value 30 days prior. Only BCS values that are not a continuation of more than 20 days are considered in the calculation.

Query 3. Find each cow from a farm site, that is not lactating more than 30 days and whose milk yield for a specific day was below two times the standard deviation from the milk yield of all cows not lactating for more than 30 days from this farm site.

Query 4. Query for each farm site the number of calvings per month and how many of the cows where diagnosed with cetosis in the following month.

Based on the textual definition of the queries we created SQL queries. To better understand the query examples, we split them into parts, which resulted in the creation of subqueries. Each of the example queries did assemble out of multiple subqueries, which partly referenced each other. At the end of this first step we could see that each of these example queries followed a similar pattern. All of the examples included two subqueries which were joined for comparison in some way. We call this two subqueries sets. One set is the "Set of interest", the other the "Set of comparison" which it is compared to.

This two sets and that they are joined for the final result was common to all example queries. The queries only differed in a few aspects. The first aspect was if the data of the set came

from the same or from different cubes. The second one was how similar the slice conditions and used dimension levels of the two cubes were, e.g., if they were grouped by the keys of the same dimension level. The third difference was how the two sets were joined. They either joined by their equal dimension levels or by a given join condition. Based on this observations we defined four different types of patterns, which are described in the following.

5.2.1 Types of Patterns

All patterns we defined based on the example queries are some sort of comparison, but they can be separated into two groups, homogenous and heterogenous comparisons. In homogenous comparisons the set of iInterest and the set of comparison use data from the same fact table. In heterogenous comparisons the two sets are based on different fact tables. In addition to this groups we defined a non-comparative pattern for simple selects. At first we are going to discuss the three types of homogenous comparisons.

Set-Base-Comparison. The set-base-comparison is derived from Query 1. In this comparison the set of interest is a subset of the set of comparison. The data for both sets is selected from the same fact table with the same slice conditions. Both sets use the same dimensions, dimension levels and the same measures, however, the set of interest has additional slice conditions and is therefore only a subset of the set of comparison.

Set-Set-Comparison. This pattern is derived from Query 2. In this comparison the set of interest and set of comparison use the same measures, dimensions and dimension levels, but they have different slice conditions. After the sets are joined further slice conditions are applied and new measures are created combining measures of both sets.

Set-Superset-Comparison. Used in Query 3, the set-superset-comparison is the loosest of the homogenous comparisons. The set of interest and set of comparison only have in common that they select data from the same fact table and use at least one common dimension level, which is necessary to join them. Other dimensions, dimension levels, measures and slice condition may differ between the two sets. After the sets are joined final slice conditions are applied.

These three patterns cover homogenous comparisons, in addition there is the heterogenous comparison, that covers the comparison of data in different fact tables. In addition to the comparison patterns we developed a non-comparative pattern to cover simple selections from a single table. Both patterns are described bellow.

Heterogenous-Comparison. This pattern is used in Query 4 to compare data of two different fact tables. Set of interest and set of comparison are completely different within this pattern. Each of them selects data from a own fact table, uses own measures, slice conditions, dimensions, and dimension levels. To join the sets a correlation between the cubes has to be stated.

Non-Comparative-Pattern. The non-comparative pattern, is used for simple queries within one table. A part from that the pattern includes slice conditions, dimensions, dimension levels and measures.

The patterns developed until now already provide the capability to execute the example queries. However they are only present as SQL, which is not very flexible in regards to new queries. Therefore, we developed a language for pattern expression, which is described in the next section.

5.2.2 A Language for Pattern Expressions

The patterns we developed, originate in the example queries which where defined by domain experts and already cover a wide range of possible queries. But as they only derive from examples they very likely do not cover all queries which will be executed in the sDWH. To avoid restrictions in the sDWH we wanted to be able to define further patterns. Moreover we needed a way to describe the query patterns, so it is possible to use them with other queries which fit the patterns.

To create such a language we had a look at the SQL queries of our patterns. In this process we cut out the specific parts of each query, e.g., the names of cubes and dimensions and created a query with blanks by doing so. While doing this with all our query patterns, similar queries with blanks emerged. Always containing bits of SQL code and blanks in between which could be filled by, e.g., fact tables, measure or attributes. Based on these examples, we created general elements, as basis for our language.

Using the parser generator ANTLR[11] we formed a grammar for this language. We choose ANTLR as it is well documented and is able to generate a Java parser for the defined grammar. As the language is rather simple at this point it could very likely be implemented in any desired parser generator. Using the defined language a pattern expression is defined. Which can be parsed to an SQL query, when appropriate data filling the elements between the SQL code snippets is given.

Listing 19 shows the pattern expression of the set-base-comparison. In the expression all SQL code snippets are enclosed in double quotation marks. This parts of the expression will remain the same during processing and will be part of the final SQL query without the quotation marks.

Listing 19: Pattern Expression of the Set-Base-Comparision

```
"WITH base AS
    (SELECT *"
    "FROM" !E <Base> !E
    "WHERE" !CL <BaseSlice> !CL ")"
"SELECT "
    !CL <dimensionLevel> !CL ","
    !CL "Base_"!+<Measure> !CL ","
    !CL "SI_"!+ <Measure> !CL
    "FROM "
        "(SELECT"
            !CL <dimensionLevel> !CL ","
            !CL ![ ^<Measure> AS "Base_"!+ !] !CL
        "FROM base"
```

[11] http://www.antlr.org/

```
        !NJL <dimension> !NJL
        "GROUP BY" !CL <dimensionLevel> !CL
        ")"
    "NATURAL JOIN"
        "(SELECT"
            !CL <dimensionLevel> !CL ","
            !CL ![ ^<Measure> AS "SI_"!+!] !CL
        "FROM base"
        !NJL <dimension> !NJL
        "WHERE" !AL ^<SIslice> !AL
        "GROUP BY" !CL <dimensionLevel> !CL
        ")"
```

All parts of the expression starting with an exclamation mark can be seen as operators. Each of these operators contains elements of the query. Depending on the operator the elements appear in different ways within the final SQL query. The `!E` operator simply adds the element to the query. The `!CL` operator is used to create comma separated lists of elements within the SQL query. The `!AL` operator has a similar behaviour but uses the `AND` keyword as separator. The `!NJL` operator creates joins within the SQL query. The last operator is the `![` operator. It is the only operator that can be nested within other operators. It is used to rename elements of the query. A more detailed description of the operator results is given in subsection 5.5.

Besides operators and SQL code there are the elements of the query. They are enclosed by angled brackets and can be seen as the blank spots in the pattern. These elements are filled with fact tables, dimension tables, measures, etc. when the pattern is parsed. If the definition of a query element starts with a circumflex, the expression behind the element will be inserted instead of the name of the element.

This pattern language is based on SQL and therefore will always result in a SQL query, but the same principle could be applied for other query languages, e.g., to create a pattern language for multidimensional expressions (MDX). Our developed language can be used by database experts to describe new patterns based on SQL, but they only describes the language specific expression of a semOLAP pattern. To complete the definition of a pattern or to execute a query more information and steps are necessary. What these informations and steps are, and how queries are executed in the sDWH is described in the following section.

5.3 Defining and Using semOLAP Patterns

To define semOLAP patterns and to execute them is a process of multiple steps. Each step uses RDF as data format and web services as means of communication with the sDWH, which is where the name semantic OLAP (semOLAP) originates. Furthermore, the pattern language described in the previous section is a integral part in the definition and execution of semOLAP patterns. This pattern language only has to be known by database experts, as only experts are involved in the fist step of the process, the definition of semOLAP patterns. These patterns are afterwards used by non expert users in the creation of pattern instances to query information from the sDWH. When pattern instances are executed, the defined patterns are used within the sDWH to create SQL queries which are executed in the sDWH's relational database. This querying process and its steps

will be described in the following sections using Query 1 (Section 5.2) and the Set-Base-Comparison (Section 5.2.1) as an example.

5.3.1 Defining semOLAP Patterns

To define a semOLAP pattern the semOLAP RDF vocabulary, with prefix `pl`[12], is needed. It is used to create a pattern definition in RDF format which includes pattern elements, the definition of its output and the pattern expression. At the beginning the elements of the pattern are defined (see Listing 20). The pattern elements describe the parts used within the pattern. They can be seen as wildcards within the pattern expression which are later filled with fact tables, dimension tables, column etc., when executing a query. These elements are used within the pattern expression, but they are not specific to a pattern expression, they are general elements that define the parts of a pattern. Furthermore, they are intended for later integrity checks, which are not implemented yet, but already considered in the vocabulary.

To define an element the RDF type `pl:PatternElement` is used. Each pattern element has two mandatory properties, `rdfs:range` and `pl:multiplicity`. Range defines which types of objects are allowed to be filled in as value in a pattern instance. Multiplicity defines how many values can be filled in for the element when creating a pattern instance. The possible options for this property are `pl:One` or `pl:OneOrMore`. The third property, `pl:partOf` is only necessary for pattern elements which are not of range `pl:FactTable`. Each pattern element with range `pl:FactTable` is filled with a table that represents the fact table of a cube, on execution of a query. All other PatternElements in some way address parts of this cube. Therefore, all of these elements have to reference a pattern element with range fact table, to indicate which cube they reference, so integrity checks can be later performed on these elements. As some pattern elements are used to define join conditions between cubes, they may reference two different cubes and therefore have two values defined for the property `pl:partOf`.

Listing 20: RDF Definition of Pattern Elements from Set-Base-Comparison

```
olap:base a pl:PatternElement;
    rdfs:range pl:ROLAPFactTable;
    pl:multiplicity pl:One.

olap:baseSlice a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:dimensionLevel a pl:PatternElement;
    rdfs:range pl:ROLAPDimensionLevel;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:dimension a pl:PatternElement;
    rdfs:range
        pl:ROLAPDimensionTable, pl:JoinElement;
```

[12] `http://dke.jku.at/semOLAPPatternLanguage#`

```
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:measure a pl:PatternElement;
    rdfs:range
        pl:CalculatedMeasureProperty, pl:ROLAPMeasure;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:siSlice a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.
```

Using the defined elements, the semOLAP pattern itself is defined as RDF type `pl:Pattern` (see Listing 21). The pattern references all its elements with the property `pl:hasElement`. In addition to a semOLAP pattern's elements, it has to be defined which of the pattern elements are the pattern's output.The elements which are part of the patterns output are referenced using the property `pl:result`. If the result element does not have its original name in the query result, its new name has to be defined. If the output element gets a prefix this is stated using the `pl:elementPrefix` property. If the element has got a different name, the new name is stated by using the property `pl:elementNewName`. The elements and the output is a general definition of the pattern.

The pattern expression which also has to be added is specific to the lanuage it is defined in. It is added using the `pl:patternExpression` property. All pattern elements associated to the semOLAP pattern are used in the pattern expression. Currently the expression is written in the SQL based language described in Section 5.2.2, but if an according language is developed and a parser is written this expression might also be exchanged, but even if the expression is written in another language all pattern elements and the defined result could stay the same.

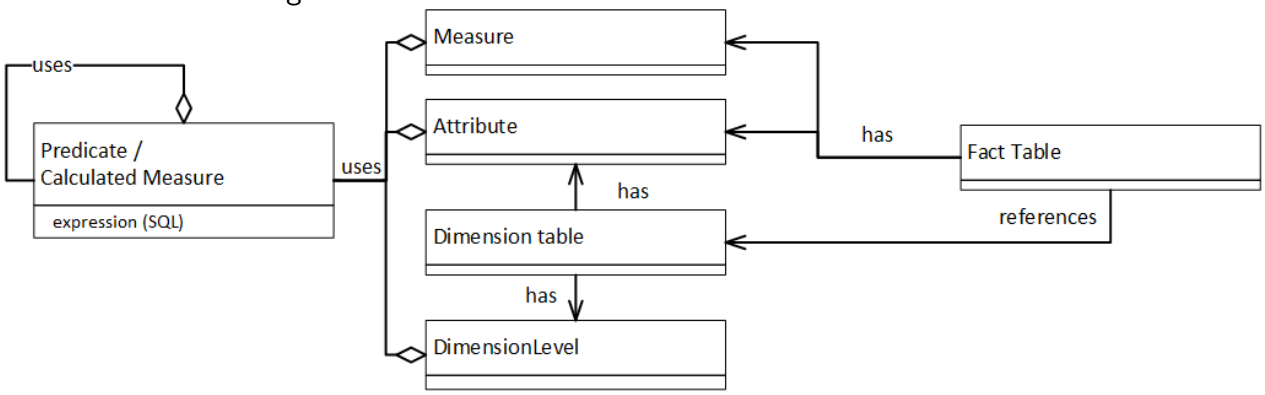
Listing 21: RDF Definition of semOLAP Pattern Set-Base-Comparison

```
olap:SetBaseComparison a pl:Pattern;
    pl:result [
        pl:element olap:measure;
        pl:elementPrefix "Base_"];
    pl:result [
        pl:element olap:measure;
        pl:elementPrefix "SI_"];
    pl:result olap:dimensionLevel;
    pl:hasElement
        olap:base, olap:baseSlice, olap:dimensionLevel,
        olap:dimension, olap:measure, olap:siSlice;
    pl:patternExpression
        '"WITH base AS (
            SELECT *"
            "FROM" !E <Base> !E
            "WHERE" !CL <BaseSlice> !CL ")"
        "SELECT " !CL <dimensionLevel> !CL ","
```

```
        !CL "Base_"!+<Measure> !CL ","
        !CL "SI_"!+ <Measure> !CL
    "FROM (
        SELECT"
            !CL <dimensionLevel> !CL ","
            !CL ![ ^<Measure> AS "Base_"!+ !]
            !CL "FROM base"
        !NJL <dimension> !NJL "GROUP BY"
        !CL <dimensionLevel> !CL ")"
    "NATURAL JOIN (
        SELECT"
            !CL <dimensionLevel> !CL ","
            !CL ![ ^<Measure> AS "SI_"!+!]
            !CL
        "FROM base"
        !NJL <dimension> !NJL
        "WHERE" !AL ^<SIslice> !AL
        "GROUP BY" KL <dimensionLevel> !CL ")"'.
```

To later use a semOLAP pattern users define values for all its pattern elements. Depending on the range of the pattern element values are for instance parts of the relational schema like fact tables, dimension tables, dimension levels or measures. In addition to the static element of the relational schema, also dynamic elements can be used as values. These dynamic elements have to be predefined by expert users, as they include SQL expressions which require more database knowledge. There are two types of dynamic elements, predicates and calculated measures. Predicates are used to define conditions. Calculated measures define new measures which are derived from existing measures or attributes in the database. The structure of a predicate or calculated measures can be seen in Figure 2.

Figure 2: Structure of a Calculated Measure or Predicate



The definition of both has to be given in RDF format as seen in Listing 22 using the types `semolap:Predicate` and `pl:CalculatedMeasure`. Both have an expression which is used within the query, which is defined using the `pl:expression` property. In addition to the expression, all elements that are necessary for the predicate/calculated measure to be executed, can be specified by the `pl:uses` property. These definitions could be used in integrity checks, which are not implemented yet.

Listing 22: Definition of Calculated Measures and Predicates

```
olap:Animal_Count a pl:CalculatedMeasureProperty;
    pl:expression "COUNT(NationalId)";
    pl:uses "NationalId".

olap:HighFat_DHIA a pl:Predicate;
    pl:expression "FatContent_DHIA > 5";
    pl:uses "FatContent_DHIA".

olap:LactationLastMonth a pl:Predicate;
    pl:expression "DayOfLactation < 30";
    pl:uses "DayOfLactation".

olap:DHIA_Fat_CT_under_30 a pl:Predicate;
    pl:expression "Milk_DHIA_FatContent_CT < 30";
    pl:uses "Milk_DHIA_FatContent_CT".

olap:20160824 a pl:Predicate;
    pl:expression "Date_='24.08.2016'";
    pl:uses "Date_".
```

With the use of predicates and calculated measures on the one hand, querying gets easier for users, as they only need to know the name and meaning of the element but not the expression behind it. On the other hand, it provides a global definition for business terms through predicates, and key process indicators through calculated measures, which can be clearly identified through all queries and results. The semOLAP patterns, predicates and calculated measures are the base of querying and are defined by database experts, but the query itself can be defined by any user. This step is described in the next section.

5.3.2 Specification of Pattern-Based Queries

The queries within the sDWH are seen as instances of the defined patterns. A query uses a pattern and provides values for all its elements. Based on the values provided, and the underlying pattern, a SQL statement is generated to query the data. Before execution the query has to be defined and loaded into the sDWH. A query is defined in RDF format using the pl vocabulary and the defined semOLAP patterns.

The query itself can be defined as `pl:ROLAPPatternInstance`, i.e., a pattern instance with relational elements as values for pattern elments. The definition of a ROLAP pattern instance for Query 1 (see Section 5.2) can be seen in Listing 23. The definition starts with the type `semolap:ROLAPPatternInstance`. The last part of the element's IRI is by convention also the name of the pattern instance, that is needed on execution. Each pattern instance has to reference a pattern using the `semolap:instanceOf` property, to state which semOLAP pattern it instantiates. Furthermore, all pattern elements which are defined in the underlying pattern, have to be used at least once. If the pattern element has got a multiplicity of `OneOrMore` it may be used more than once.

Listing 23: semOLAP Definition of Query 1 from Section 5.2

```
olap:Comparison_HighFat a pl:ROLAPPatternInstance;
    pl:instanceOf olap:SetBaseComparison;
    olap:base "Milk";
    olap:baseSlice olap:20160824;
    olap:dimensionLevel "FarmSiteId";
    olap:dimension "FarmSite";
    olap:measure olap:Animal_Count;
    olap:siSlice olap:HighFat_DHIA;
    olap:siSlice olap:LactationLastMonth;
    olap:siSlice olap:DHIA_Fat_CT_under_30.
```

In addition to using predicates, calculated measures, and strings as values for pattern elements, join elements can be used as well. If the value of an element is a dimension table, for instance the element dimension in Listing 23, the table can also be referenced as a join element, which gives additional information if this element is used in a join (see Listing 24). The join element is defined using a blank node. Within this blank node is at least the property `pl:table` stating the table that is joined. There are three more properties to use, but they are only optional. The first is the `pl:condition` property. It is used to specify a join condition. If a join condition is given the table will be joined with condition instead of a natural join. The second is the property `pl:join`. It defines the type of join. By default, the tables are joined by inner join. Using the join property this can be changed to other join types like "LEFT" or "RIGHT". The last possible property is `pl:order`. It defines in which order the tables are joined. The lowest possible order is `1`. The order has to be unique among the join elements of a ROLAP pattern instance. How the joins are created can be seen in Section 5.5.

Listing 24: Definition of a Join Element

```
olap:Comparison_HighFat a pl:ROLAPPatternInstance;
    ....
    olap:dimension "FarmSite",
        [pl:table "Enterprise";
         pl:join "LEFT";
         pl:condition
         "FarmSite.FarmSiteId=
          Enterprise.FarmSiteId";
         pl:order 1];
    ...
```

An optional property to a pattern instance is the `semolap:persistAs` property. It has the possible values `pl:View` and `pl:Snapshot`. If it has got the value `pl:View`, a view with the query behind the pattern instance is created in the sDWH's database, when the pattern instance is loaded into the sDWH. If it has got the value `pl:Snapshot` a table based on the query behind the pattern instance is created in the sDWH's database, when the pattern instance is loaded into the sDWH. This table contains the data of the query result. After the view or snapshot is created, it is possible to use the pattern instance like a table in other pattern instances. If this is the case, the pattern instances reference the view or snapshot in the relational database, to query data from the underlying pattern instance.

This concludes the definition of the ROLAP pattern instances, but for the semOLAP patterns and ROLAP pattern instances to be used they have to be loaded into the sDWH, how this is done is described in the next section.

5.4 Web Service Interface for Querying the sDWH

To add the definitions of semOLAP patterns, ROLAP pattern instances, calculated measures and predicates to the sDWH, RESTful web services are used. Before a pattern instance can be executed, or a semOLAP pattern, calculated measure, or predicate used it has to be loaded into the sDWH. There are three web services to manage this a service to insert, one to delete and one to do an insert replace. Each of the web services is called with a HTTP POST request including data in RDF format. It is possible to define a semOLAP pattern in the same data it is used the first time, as long as the definition is complete.

To insert new definitions the insert web service is called using the URL `<server>/AgriProKnowDBService/rest/PatternService/InsertPatternData`. On insert of the data, several checks are done. The first is to ensure that there are no objects with duplicate names when the pattern data are added. Furthermore, if the new data include `pl:Pattern` elements they are checked for a valid pattern expression and that at least one result element is defined. If pattern instances are added to the sDWH a check occurs to ensure that a valid semOLAP pattern is referenced. If the pattern instance is to be persisted, it is executed and a view or snapshot created. If any error during these checks occur the HTTP status code 500 (server error) and an according error message are returned, without adding the data to the sDWH. If the data were successfully added they are stored in the sDWH's triple store.

The web service for deletion of pattern data only needs elements of the types `pl:Pattern`, `pl:ROLAPPatternInstance`, `pl:CalculatedMeasureProperty` and `pl:Predicate`. Only the IRI and type of an element has to be stated. All triples with the stated elements as subject are deleted from the sDWH, but no integrity checks on the use of the deleted elements are performed. Therefore, the deletion of elements should be used with care, as it may lead to errors on execution if a needed element was deleted. If a pattern instance is deleted which was persisted as view or snapshot in the relational database, the view or table is deleted as well. The web service for deletion is called using the URL `<server>/AgriProKnowDBService/rest/PatternService/DeletePatternData`

The last web service is the insert replace service, it is a combination of the web services described above. It has the exact same result as calling the delete pattern data service first, and then calling the insert pattern data service with the same data. First, all semOLAP patterns, pattern instances, calculated measure and predicate stated in the data sent to the web service are delete from the triple store. Afterwards the insert web service is called to add the new data. Therefore all definition of elements have to be defined in the same way as if they were to be sent to the insert web service. The insert replace web service is called using the URL `<server>/AgriProKnowDBService/rest/PatternService/InsertReplacePatternData`

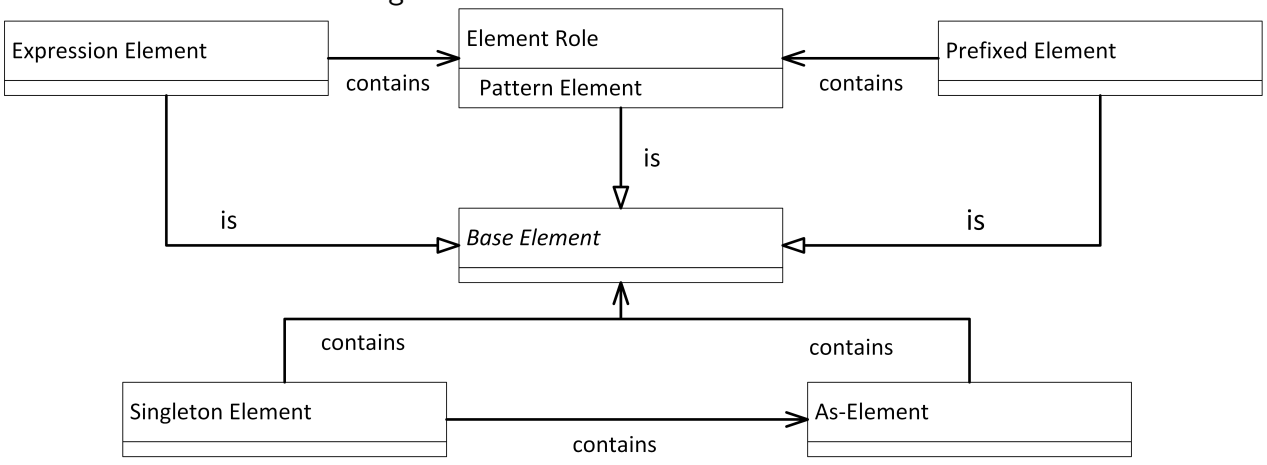
5.5 Pattern Instance Execution

The execution of a pattern instance is done by the Pattern Engine. The exeuction is started by calling the web service of the sDWH with the URL `<server>/AgriProKnowDBService/-rest/Execute/`. At the end of the url the name of the pattern instance to execute has to be added. On start of the execution the ROLAP pattern instance and the according semOLAP pattern are loaded from the triple store into Java objects.

Using these Java objects the pattern engine which is based on a parser generated by ANTLR start the generation of the SQL statement. Basically the pattern engine works through the pattern expression and fills it with the values stated in the pattern instance. The pattern expression consists of multiple parts.

The simplest parts of the pattern expression are the three types of base elements, which are, element role, expression element and prefixed element (see Figure 3). If a the name of a pattern element is given in angled brackets (<...>) it is an element role. An element role is replaced by the value stated for the pattern element in the pattern instance. All other elements and operators are based on the element role. The expression element is described by an element role with a leading circumflex (^). It is important for predicates and calculated measures, as it represents the expression behind the value set in the pattern instance, e.g., if the value of a pattern element within an expression element is a predicate, the expression element is replaced by the expression of the predicate instead of its name. If the value of the pattern element does not have an expression, its name is inserted instead. The third base element is the prefixed element. It is used to add a prefix to an element role. A prefix is defined by a string in double quotations with the operator `!+` at the end. If a prefix is written before an element role, a prefixed element is created.

Figure 3: Base Elements and As-Element

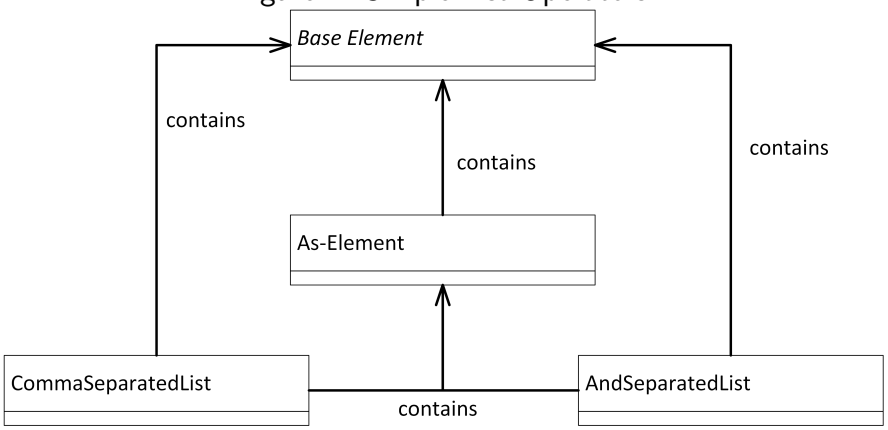


These base elements are used in the as-element operator. The as-element is enclosed by the symbols `![`. It is used to define an SQL as-clause, which renames a column. The as-element consists of a base element, the key word "AS", and an optional second part which is either a string or a prefix. During parsing the first part is replaced with the result of the parsing of the base element. If the second part is empty it is replaced by the result of the element role of the base element in the first part, e.g., the value of the pattern element. If the second part is a prefix, a prefixed element is created with the defined prefix and the element role of the base element. If the second part is a string, the string is added behind the "AS" keyword, giving the base element a new name. To only have a string in the second element only works, if there is just one value

for the pattern element stated in the base element of the first part, as otherwise multiple columns with the same name would be created. To define that a base element only consists of one value it is put within a singleton element. The singleton element is enclosed by the symbol `!E`.

If a pattern element can have multiple values it needs to be enclosed in a list operator. The two simplest list operators are the comma separated list and the and separated list (see Figure 4). Both use a single base element or as-element. The comma separated list is defined by enclosing this element in the symbol `!CL`. While parsing, the values behind the enclosed element are put alongside each other separated by a comma (,). The and list symbol is `!AL` and its separator the keyword "AND". Both operators only add the separator between values, which means that there is no separator at the end of the list, even if there is only one value in the list.

Figure 4: Simple List Operators

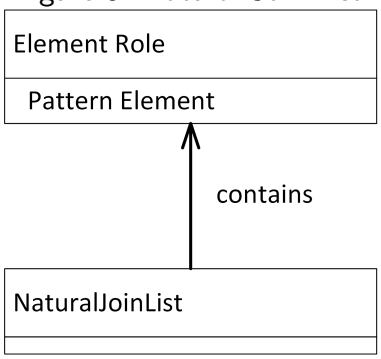


Another more complex list operator is the natural join list. The natural join list is defined by enclosing an element role (no other base element) (see Figure 5) with the symbol `!NJL`. The natural join list is used to connect values of pattern elements which represent tables, e.g., fact or dimension tables. The natural join list also deals with the special attributes of join elements. In any other case join elements are only treated like an element with its value defined by the property `pl:table` and its expression by `pl:condition`. If used for an element role within a natural join list, join elements give additional specification for the join. By default the values in a join list are joined by natural inner join, this can be changed when using a join element. When parsing the values of a natural join list, the values defined as join elements are processed first. They are processed in ascending order according to their `pl:order` properties. The join elements without order are processed after the ordered ones. If a condition is defined in the join element an on-clause with this condition is created, and the "NATURAL" keyword of this join removed. Furthermore, if a join type (property `pl:join`) is specified e.g. "LEFT", "RIGHT" it is added to the join of the element. After all join elements are processed, all other values are processed. Each of them is simply added as natural inner join at the end of the join list.

By processing the elements and operators of the pattern expression with the values of a pattern instance a SQL query is created. An example of such a statement, after processing the set-base-comparison pattern (see Listing 21, p. 47) and the ROLAP pattern instance seen in Listing 23 (p. 49) can be seen in Listing 25.

The generated SQL statement is then executed within the database of the sDWH. The result is exported in CSV(;) format. The first line of the resulting CSV is used as header and states the column names within the result. In the future additional schema information queried from the

Figure 5: Natural Join List



Listing 25: Generated Statement for Set-Base-Comparison Pattern of ROLAP pattern instance described in Listing 23

```
WITH base AS
    (SELECT *
    FROM Milk
    WHERE Date_='24.08.2016' )
SELECT
    FarmSiteId,
    Base_Animal_Count,
    SI_Animal_Count
    FROM
        (SELECT
            FarmSiteId,
            COUNT(NationalId) AS Base_Animal_Count
        FROM base
        NATURAL JOIN FarmSite
        GROUP BY FarmSiteId
        )
    NATURAL JOIN
        (SELECT
            FarmSiteId,
            COUNT(NationalId) AS SI_Animal_Count
        FROM base
        NATURAL JOIN FarmSite
        WHERE
            FatContent_DHIA > 5 AND
            DayOfLactation < 30 AND
            Milk_DHIA_FatContent_CT < 30
        GROUP BY FarmSiteId
        )
```

sDWH's RDF schema could be returned as well, to add a better description of the data. Another possibility would be the use of R2RML [7], the W3C's RDB to RDF Mapping language to export the structure of the result as well as the data of the result in RDF format. The exported RDF data could therefore be exported alongside the RDF definition of the ROLAP pattern instance it came from. This would provide a description for the data's origin, e.g., where a calculated measures was derived from.

5.6 Querying at the qb Level

We refer to the formulation of pattern-based queries as pattern instances. Two options for pattern instances exist which differ in their level of conceptualisation. The first option, which is the main focus in this thesis, employs star schema elements and is referred to as ROLAP pattern instance. The second option, of which this thesis presents only a preliminary solution, employs qb and qb4o elements and is referred to as qb pattern instance. In the following we briefly present the qb level and its current state of implementation.

Defining queries based on qb elements has several advantages. If query creation is done with qb schema elements instead of relational elements of the database, a further abstraction from SQL occurs. The user does not need to have knowledge about the relational database of the sDWH and its schema, the user only needs to know the RDF schema of the sDWH, which would make the user's knowledge independent from the way the data are stored in the sDWH.

Another advantage of querying at the qb level is the easier export of a query result in RDF format using qb, qb4o and qbgen vocabularies. The semOLAP pattern defines which of its pattern elements are part of the result. As the pattern elements are filled with qb schema elements in the qb pattern instance, and all of the qb schema elements are somehow part of a cube, the result of the query can also be seen as cube and a corresponding qb definition of the result can be created and linked to the pattern instance that was used to derive the result cube. The formulation of qb pattern instances preserves the derivation chain of derived cubes and consequently facilitates interpretation and reproducibility of analyses.

There are also several challenges associated with query generation from qb pattern instance. The definition of calculated measures and predicates contains an expression which is specific to the target system. The definition of the expression in a target language such as SQL lowers the barrier for database experts to use the sDWH at the expense of general applicability. Nevertheless, if expressions in multiple target languages are provided for calculated measures and predicates, pattern instances that use these calculated measures and predicates can be translated into queries for different platforms.

Qb, qb4o and qbgen only describe schema and data. The qb pattern instances need to be converted into a query language. As the AgriProKnow sDWH is based on a relational database, the qb pattern instances have to be converted to SQL for execution. The current prototype translates qb pattern instances into ROLAP pattern instances. ROLAP pattern instance can then be executed using the capabilities described in previous sections. The result of the execution of a ROLAP pattern instance is a table. The result table is then enriched with qb and qb4o data. The enrichment of the result table with qb and qb4o data requires a mapping between relational schema and qb/qb4o elements. The current implementation derives a mapping from the relationship between ROLAP pattern instance and qb pattern instance.

5.6.1 Defining a qb Pattern Instance Using RDF

Qb pattern instances, like ROLAP pattern instances, instantiate semOLAP patterns. A semOLAP pattern has two main parts, the definition of pattern elements and the pattern expression. The qb pattern instance only depends on the definition of elements, as the pattern expression, which is specific to the target language, only comes into play when a ROLAP pattern instance is executed. The definition of a qb pattern instance is done in the same way as a ROLAP pattern instance, the only difference being that all elements in the semOLAP pattern are filled with elements from qb, qb4o and qbgen as values, instead of relational elements. Listing 26 shows an example qb pattern instance of Query 1 from Section 5.2.

Listing 26: Definition of a qb Pattern Instance

```
olap:Comp_HighFat a pl:QbPatternInstance;
    pl:instanceOf olap:SetBaseComparison;
    olap:base agri:Milk;
    olap:baseSlice olap:20160824;
    olap:dimensionLevel agri:FarmSite;
    olap:dimension agri:FarmSiteDim;
    olap:measure olap:Animal_Count;
    olap:siSlice olap:HighFat_DHIA;
    olap:siSlice olap:LactationLastMonth;
    olap:siSlice olap:DHIA_Fat_CT_under_30.
```

The qb pattern instance in Listing 26 corresponds to the ROLAP pattern instance in Listing 23, Section 5.3.2. The referenced predicates and calculated measures are the same, albeit at different levels of conceptualisation. For example, the value for `olap:dimensionLevel` is now referencing the `qb4o:LevelProperty` defining the farm site level instead of the column `NationalId` representing the level in the relational schema. Also the pattern element `olap:dimension` now references the definition of the farm site dimension instead of the dimension table `FarmSite`.

5.6.2 Mapping qb Pattern Instances to ROLAP Pattern Instances

We provide a mapping from qb pattern instances to ROLAP pattern instances. This mapping relies on the type that a value for a pattern element within the qb pattern instance has. For each type a specific mapping is defined. As the same predicates and calculated measures are used on qb and ROLAP level, they are not mapped but only passed on to the ROLAP pattern instance.

qb4o:LevelPorperty. A dimension level defined by a `qb4o:LevelProperty` is identified by its key attributes. Therefore a level property is mapped to its key columns, e.g., `agri:FarmSite` is mapped to `NationalId`.

qb:DataStructureDefiniton and Analysis Table. Elements that are of type `qb:DataStructureDefinition` or an analysis table are representing a fact table and are therefore mapped to the name of the fact table, which by convention is the IRI of the object without its prefix. Therefore, the analysis table `agri:Milk` is mapped to `Milk`.

qb:DimensionProperty. Dimensions represented by a `qb:DimensionProperty` require are more complex mapping. Hereby, the implementation type of the dimension is of essence. If the dimension is degenerate, no mapping at all occurs and the dimension is not passed on to the ROLAP pattern instance, as the dimension is stored within the fact table, the fact table (analysis table or data structure definition) itself hast to be stated, not the dimension it contains. If the dimension is of type star, it is stored in one dimension table. Therefore, it is mapped to the name of its dimension table, which by convention is the IRI of its root element without prefix. If the dimension is of type snowflake it is scattered across multiple dimension tables which are in hierarchical order. Therefore, the levels of the dimension are ordered ascending beginning from the root level and each level is mapped to a join element. The `pl:table` property of each join element is by convention the IRI of the level it represents without its prefix.

qbgen:ComplexAttribute Complex attributes are also added as join elements. As they only describe additional information, the join element for the complex attribute is defined with join type "LEFT". So the join does not restrict the other data. Furthermore, a join element for a complex attributes is the last in order, if there are multiple join elements as values of a pattern element.

5.6.3 Returning the Result of a qb Pattern Instance

Based on the previously defined implicit mapping rules a ROLAP pattern instance is created out of a qb pattern instance. The created ROLAP pattern is mapped to an SQL query, which is then executed on the sDWH's database. The query result is exported as RDF. At first the structure of the query result is exported. Therefore, the definition of the qb pattern instance is exported and a `qb:DataStructureDefinition` specifying the result cube is generated, as the result pattern elements defined in the semOLAP pattern are filled with qb elements, the values are dimension levels or measures which define a new cube. Note that for complex attributes it is not possible to be value of a result pattern element in the current version of the prototype. Furthermore, at the moment, it is not possible to include two dimension levels from the same dimension as values for result pattern elements. To link to the pattern instance, the data structure definition has a `semolap:resultOf` property, referencing the qb pattern instance.

After the export of structure, the result data are exported. Each row of the query result is added as `qb:Observation` to the export of the result. After all rows are exported, all referenced dimension entities are exported as `qb4o:LevelMembers`. In addition to the dimension entities directly referenced, the roll-ups of these entities are exported as well, which concludes the export of the query result.

6 Summary and Future Work

This thesis presents a first prototype of the AgriProKnow project's semantic data warehouse (sDWH) using a combination of semantic technologies and relational database technology. The sDWH allows for schema creation and modification since the AgriProKnow project is still ongoing. Semantic technologies assist designers and end users in their interactions with the sDWH. Most interaction with the sDWH is based on qb, qb4o and qbgen, RDF vocabularies for the representation of multidimensional models. Based on the RDF definition of the data model the sDWH creates a relational database schema for storage of the instance data. Using semOLAP patterns, analysts may query the sDWH.

In this thesis we present a first prototype of the sDWH of the AgriProKnow project which is still ongoing. Future work will include the following:

- **Additional query patterns and extensions for pattern language:** The query patterns presented in this thesis were developed based on the requirements expressed by domain experts involved in the AgriProKnow project. Although applicable to other use cases the identified query patterns are not exhaustive. Using the pattern language presented in this thesis, additional patterns may be defined for the AgriProKnow project and other use cases. Furthermore, more powerful query patterns may demand extensions of the pattern language.
- **Explicit mapping from qb to relational schema:** The current mapping from qb to relational schema follows a convention that is implemented in the program code and informally described in this thesis. Future work will provide an explicit mapping from qb to relational schema, possibly in a declarative language. Thus, the mapping will not be hidden in program code and may facilitate conversion of relational query results into qb for ease of interpretation.
- **Graphical user interface (GUI) for pattern-based querying:** Query patterns serve as the fundamental for the provision of intuitive query facilities by the sDWH. Query patterns alone, however, are probably insufficient for non-experts in database technology. Therefore, future work will provide a GUI based on the query patterns which could also incorporate aides for formulating queries such as integrity checks and code completion.
- **Rule-based farm operations:** Some of the analyses may be run periodically, e.g., once a day, in order to react automatically to detected risk situations, effectively leading to the development of an active sDWH. In this active sDWH, event-condition-action (ECA) rules model data-driven behaviour. Risk situations are defined using semOLAP patterns and constitute events and conditions of ECA rules.
- **Performance:** The current state of sDWH developments focuses on data models, update procedures and query models. Performance issues are disregarded. Future work will provide optimizations for loading procedures, such as non-RDF based (not only RDF-based) loading of instance data with bulk loading of CSV files, definition of index structures and materializations of aggregate views, and improved definitions of queries.

References

[1] Marcelo Arenas et al. *A Direct Mapping of Relational Data to RDF6: W3C Recommendation*. Ed. by World Wide Web Consortium. 2012. URL: https://www.w3.org/TR/rdb-direct-mapping/.

[2] Ladjel Bellatreche and Mukesh K. Mohania, eds. *Data Warehousing and Knowledge Discovery*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014. ISBN: 978-3-319-10159-0. DOI: 10.1007/978-3-319-10160-6.

[3] Clemente Borges and José Macías. "Facilitating the interaction with data warehouse schemas through a visual web-based approach". In: *Computer Science and Information Systems* 11.2 (2014), pp. 481–501. ISSN: 1820-0214. DOI: 10.2298/CSIS131130032B.

[4] Guntars Bumans. "Relational Database information availability to Semantic Web technologies". PhD thesis.

[5] Caterina Caracciolo et al. "The AGROVOC linked dataset". In: *Semantic Web* 4.3 (2013), pp. 341–348. ISSN: 1570-0844.

[6] Richard Cyganiak, Dave Reynolds, and Jeni Tennison. *The RDF Data Cube Vocabulary: W3C Recommendation*. Ed. by World Wide Web Consortium. 2014. URL: https://www.w3.org/TR/vocab-data-cube/.

[7] Souripriya Das, Seema Sundara, and Richard Cyganiak. *R2RML: RDB to RDF Mapping Language: W3C Recommendation*. Ed. by World Wide Web Consortium. 2013. URL: https://www.w3.org/TR/r2rml/.

[8] Sami El-Mahgary and Eljas Soisalon-Soininen. "A form-based query interface for complex queries". In: *Journal of Visual Languages & Computing* 29 (2015), pp. 15–53. ISSN: 1045926X. DOI: 10.1016/j.jvlc.2015.03.001.

[9] Lorena Etcheverry and Alejandro A. Vaisman. "QB4OLAP: A New Vocabulary for OLAP Cubes on the Semantic Web". In: *Proceedings of the Third International Conference on Consuming Linked Data - Volume 905*. COLD'12. Aachen, Germany: CEUR-WS.org, 2012, pp. 27–38. URL: http://dl.acm.org/citation.cfm?id=2887367.2887370.

[10] Lorena Etcheverry, Alejandro Vaisman, and Esteban Zimányi. "Modeling and Querying Data Warehouses on the Semantic Web Using QB4OLAP". In: *Data Warehousing and Knowledge Discovery*. Ed. by Ladjel Bellatreche and Mukesh K. Mohania. Vol. 8646. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 45–56. ISBN: 978-3-319-10159-0. DOI: 10.1007/978-3-319-10160-6{\textunderscore}5.

[11] Saleh Ghasemi, Wo-Shun Luk, and Norah Alrayes. "M2RML: Multidimensional to RDF Mapping Language". In: *2014 25th International Workshop on Database and Expert Systems Applications (DEXA)*, pp. 263–267. DOI: 10.1109/DEXA.2014.61.

[12] Matteo Golfarelli and Stefano Rizzi. *Data warehouse design: Modern principles and methodologies*. New York: McGraw-Hill, 2009. ISBN: 978-0071610391.

[13] E. Jahanshiri and S. Walker. "Agricultural Knowledge-Based Systems at the Age of Semantic Technologies". In: *International Journal of Knowledge Engineering-IACSIT* 1.1 (2015), pp. 64–67. ISSN: 23826185. DOI: 10.7763/ijke.2015.v1.11.

[14] Ralph Kimball. *The data warehouse lifecycle toolkit: expert methods for designing, developing, and deploying data warehouses*. John Wiley & Sons, 1998.

[15] Franck Michel, Johan Montagnat, and Catherine Faron-Zucker. *A survey of RDB to RDF translation approaches and tools*. URL: https://hal.archives-ouvertes.fr/hal-00903568.

[16] Alistair Miles and Sean Bechhofer. *SKOS simple knowledge organization system reference: W3C Recommendation*. Ed. by World Wide Web Consortium. 2009. URL: https://www.w3.org/TR/skos-reference/.

[17] Thomas Neuböck et al. "Ontology-driven business intelligence for comparative data analysis". In: *Business Intelligence*. Springer, 2014, pp. 77–120.

[18] Mary Beth Roeser. *Oracle Database SQL Language Reference, 12c Release 1 (12.1)*. 2016. URL: https://docs.oracle.com/database/121/SQLRF/statements_9016.htm.

[19] Oscar Romero et al. "Describing Analytical Sessions Using a Multidimensional Algebra". In: *Data Warehousing and Knowledge Discovery*. Ed. by David Hutchison et al. Vol. 6862. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 224–239. ISBN: 978-3-642-23543-6.

[20] Catherine Roussey et al. "Ontologies in agriculture". In: *AgEng 2010, International Conference on Agricultural Engineering*. 2010.

[21] William Rowen et al. "An analysis of many-to-many relationships between fact and dimension tables in dimensional modeling". In: *Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW 2001)*. 2001.

[22] Christoph G. Schuetz et al. "Reference Modeling for Data Analysis: The BIRD Approach". In: *International Journal of Cooperative Information Systems* 02 (2016). ISSN: 0218-8430.

[23] Rick Sherman. *Business intelligence guidebook: From data integration to analytics*. Amsterdam: Elsevier, 2015. ISBN: 9780124115286.

[24] Slobodanka Dana Kathrin Tomic et al. "agriOpenLink: Semantic Services for Adaptive Processes in Livestock Farming". In: *International Conference of Agricultural Engineering*. 2014.

[25] Jovan Varga et al. "QB2OLAP: Enabling OLAP on Statistical Linked Open Data". In: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 1346–1349. DOI: 10.1109/ICDE.2016.7498341.

List of Figures

List of Tables

Listings

Appendix A The qbgen Vocabulary

```
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:      <http://www.w3.org/2002/07/owl#> .
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#> .

@prefix qb:       <http://purl.org/linked-data/cube#> .
@prefix qb4o:     <http://purl.org/qb4olap/cubes#> .
@prefix qbgen:    <http://dke.jku.at/qbgen#>.

#################################################################
#
#    Classes
#
#################################################################

qbgen:ComplexAttribute a rdfs:Class;
    rdfs:subClassOf qb4o:LevelAttribute.

qbgen:ComplexComponentSet a rdfs:Class.

qbgen:ComplexAttributeInstance a rdfs:Class.

qbgen:KeyColumnsSet a rdfs:Class.

qbgen:ImplementationTyp a rdfs:Class;
    rdfs:comment "Describes in which way a dimension should be
       implemented in SQL"@en.

qbgen:RenamingSet a rdfs:Class;
    rdfs:subClassOf qb:ComponentSet.

#################################################################
#
#    Object Properties
#
#################################################################

qbgen:hasID a rdf:Property;
    rdfs:domain qbgen:ComplexAttribute;
    rdfs:range qb:AttributeProperty;
    rdfs:comment "Indicates which columns of a ComplexAttribute
       are part of its PrimaryKey".

qbgen:linksDimension a rdf:Property;
    rdfs:domain qbgen:ComplexAttribute;
    rdfs:range qb4o:LevelProperty;
```

```
    rdfs:comment "Indicates which Dimensions are part of a
       ComplexAttribute, they are automaticly part of its primary
       key".

qbgen:hasAttribute a rdf:Property;
    rdfs:domain qbgen:ComplexAttribute;
    rdfs:range qb:ComponentProperty.

qbgen:implementation a rdf:Property;
    rdfs:domain qb:DimensionProperty;
    rdfs:range qbgen:ImplementationTyp.

qbgen:excludeFromKey a rdf:Property;
    rdfs:domain qb:ComponentSet;
    rdfs:range xsd:boolean;
    rdfs:comment "To indicate that a Dimension should not be part
        of the Cubes PrimaryKey".

qbgen:renaming a rdf:Property;
    rdfs:domain qb:DataStructureDefinition, qb4o:LevelProperty;
    rdfs:range qbgen:RenamingSet.

qbgen:rename a rdf:Property;
    rdfs:domain qbgen:RenamingSet;
    rdfs:range qb:AttributeProperty, qb4o:LevelProperty.

qbgen:renameTo a rdf:Property;
    rdfs:domain qbgen:RenamingSet;
    rdfs:range xsd:string.

qbgen:keys a rdf:Property;
    rdfs:domain qb:DataStructureDefinition, qbgen:
       ComplexAttribute;
    rdfs:domain qbgen:KeyColumnsSet.

qbgen:keyLevel a rdf:Property;
    rdfs:domain qbgen:KeyColumnsSet;
    rdfs:range qb4o:LevelProperty.

qbgen:keyColName a rdf:Property;
    rdfs:domain qbgen:KeyColumnsSet;
    rdfs:range xsd:string.

qbgen:instanceOf a rdf:Property;
    rdfs:domain qbgen:ComplexAttributeInstance;
    rdfs:range qbgen:ComplexAttribute.

qbgen:SnowFlake a qbgen:ImplementationTyp.
```

```
qbgen:Star a qbgen:ImplementationTyp.

qbgen:Degenerate a qbgen:ImplementationTyp.
```

Appendix B Grammar of the Pattern Language

```
grammar QueryPattern;

sqlTemplate : (SQLTEXT | patternCmd)+;
patternCmd : singleton | commaList | andList | njList ;

singleton : SINGELTONSEP (baseElement | asPattern) SINGELTONSEP;
commaList : COMMALISTSEP (baseElement | asPattern) COMMALISTSEP;
andList : ANDLISTSEP baseElement ANDLISTSEP;
njList : NJLISTSEP elementRole NJLISTSEP;


baseElement : elementRole
                | prefixedElementRole
                | expressionElement;
asPattern: ASPATTERNBEGIN baseElement AS (prefix | SQLTEXT)?
   ASPATTERNEND;

expressionElement : EXPRESSIONELEMENTSYMBOL elementRole;
prefixedElementRole : prefix elementRole;
prefix : SQLTEXT CONCAT;


elementRole : '<' ID '>';


EXPRESSIONELEMENTSYMBOL: '^';
SINGELTONSEP: '!E';
COMMALISTSEP: '!CL';
ANDLISTSEP: '!AL';
NJLISTSEP: '!NJL';
CONCAT: '!+';
ASPATTERNBEGIN: '![';
ASPATTERNEND: '!]';
AS: 'AS';


ID : [a-zA-Z0-9_\-]+ ;
SQLTEXT : '"' .*? '"' ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Appendix C SemOLAP Pattern Language Vocabulary

```
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:      <http://www.w3.org/2002/07/owl#> .
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#> .

@prefix qb:       <http://purl.org/linked-data/cube#> .
@prefix qb4o:     <http://purl.org/qb4olap/cubes#> .
@prefix qbgen:    <http://dke.jku.at/qbgen#>.
@prefix pl:   <http://dke.jku.at/semOLAPPatternLanguage#>.

###################################################################
#
#    Classes
#
###################################################################

pl:CalculatedMeasureProperty a rdfs:Class;
    rdfs:subClassOf qb:MeasureProperty;
    rdfs:comment "A measure calculated out of other measures"@en.

pl:Predicate a rdfs:Class;
    rdfs:comment "A restriction to be used in slice conditions"
       @en.

pl:PatternInstance a rdfs:Class;
    rdfs:comment "An executable instance of a BuildingBlock".

pl:QbPatternInstance a rdfs:Class;
    rdfs:subClassOf pl:PatternInstance.

pl:RolapPatternInstance a rdfs:Class;
    rdfs:subClassOf pl:PatternInstance.

pl:RenamedDimLevel a rdfs:Class;
    rdfs:subClassOf qb4o:LevelProperty.

pl:Pattern a rdfs:Class.

pl:AsPattern a rdfs:Class.

pl:PersistentElement a rdfs:Class.

pl:PatternElement a rdfs:Class;
    rdfs:comment "Element of a Pattern, which has to be used in
       PatternInstance".

pl:multiplicityValue a rdfs:Class.
```

```
pl:JoinElement a rdfs:Class.

pl:RolapFactTable a rdfs:Class;
    rdfs:comment "Used in Range constraints for PatternElements"
       @en.

pl:RolapDimensionTable a rdfs:Class;
    rdfs:comment "Table representing a Star Dimension or Level of
        a SnowFlake Dimension.
                  Used in Range constraints for PatternElements"
                     @en.

pl:RolapDimensionLevel a rdfs:Class;
    rdfs:comment "Id-Attribute of a DimensionLevel. Used in Range
        constraints for PatternElements"@en.

pl:RolapAttribute a rdfs:Class;
    rdfs:comment "Any describing Column. Used in Range
       constraints for PatternElements"@en.

pl:RolapMeasure a rdfs:Class;
    rdfs:comment "Used in Range constraints for PatternElements"
       @en.

pl:RolapComplexAttributeTable a rdfs:Class;
    rdfs:comment "Used in Range constraints for PatternElements"
       @en.

#################################################################
#
#    Properties
#
#################################################################

pl:expression a rdf:Property;
    rdfs:domain pl:CalculatedMeasureProperty, pl:Predicate;
    rdfs:range xsd:string;
    rdfs:comment "The SQL Expression how the measure is calulated
       ".

pl:instanceOf a rdf:Property;
    rdfs:domain pl:PatternInstance;
    rdfs:range pl:Pattern.

pl:result a rdf:Property;
    rdfs:domain pl:Pattern;
    rdfs:range pl:PatternElement, pl:AsPattern.
```

```
pl:patternExpression a rdf:Property;
    rdfs:domain pl:Pattern;
    rdfs:range xsd:string.

pl:element a rdf:Property;
    rdfs:domain pl:AsPattern;
    rdfs:range xsd:string.

pl:elementPrefix a rdf:Property;
    rdfs:domain pl:AsPattern;
    rdfs:range xsd:string.

pl:elementNewName a rdf:Property;
    rdfs:domain pl:AsPattern;
    rdfs:range xsd:string.

pl:dimension a rdf:Property;
    rdfs:domain pl:RenamedDimLevel;
    rdfs:range qb4o:LevelProperty.

pl:inFact a rdf:Property;
    rdfs:domain pl:RenamedDimLevel;
    rdfs:range qb:DataStructureDefinition.

pl:persistAs a rdf:Property;
    rdfs:domain pl:PatternInstance;
    rdfs:range pl:PersistentElement.

pl:hasElement a rdf:Property;
    rdfs:domain pl:Pattern;
    rdfs:range pl:PatternElement.

pl:multiplicity a rdf:Property;
    rdfs:domain pl:PatternElement;
    rdfs:range pl:multiplicityValue.

pl:partOf a rdf:Property;
    rdfs:domain pl:PatternElement;
    rdfs:domain pl:PatternElement.

pl:uses a rdf:Property;
    rdfs:domain pl:Predicate, pl:CalculatedMeasureProperty;
    rdfs:range pl:Predicate,  pl:CalculatedMeasureProperty, qb4o:
       LevelProperty,
                qb:MeasureProperty, qb:AttributeProperty, qbgen:
                   ComplexAttribute, xsd:string.

pl:table a rdf:Property;
    rdfs:domain pl:JoinElement;
```

```
    rdfs:comment "A table to join"@en.

pl:order a rdf:Property;
    rdfs:domain pl:JoinElement;
    rdfs:range xsd:integer;
    rdfs:comment "Specifies the order of joins"@en.

pl:condition a rdf:Property;
    rdfs:domain pl:JoinElement;
    rdfs:range xsd:string;
    rdfs:comment "Specifies a JOIN condition"@en.

pl:join a rdf:Property;
    rdfs:domain pl:JoinElement;
    rdfs:range xsd:string;
    rdfs:comment "Specifies a type of join e.g. LEFT, RIGHT"@en.

pl:View a pl:PersistentElement.
pl:Snapshot a pl:PersistentElement.

pl:One a pl:multiplicityValue.
pl:OneOrMore a pl:multiplicityValue.
```

Appendix D Definitions of semOLAP Patterns and Pattern Instances

```
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:      <http://www.w3.org/2002/07/owl#> .
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#> .

@prefix qb:       <http://purl.org/linked-data/cube#> .
@prefix qb4o:     <http://purl.org/qb4olap/cubes#> .
@prefix qbgen:    <http://dke.jku.at/qbgen#>.
@prefix olap:   <http://dke.jku.at/semOLAP#>.
@prefix agri:     <http://agriproknow.com/vocabulary/AgriPro#>.
@prefix pl:   <http://dke.jku.at/semOLAPPatternLanguage#>.


##########################
## Pattern
##########################

## NonComparative Pattern

olap:NonComparative a pl:Pattern;
    pl:result olap:resultMeasure;
    pl:result olap:dimensionLevel;
    pl:hasElement olap:resultMeasure, olap:base, olap:slice, olap
       :dimensionLevel, olap:dimension, olap:having;
    pl:patternExpression '"SELECT" !CL <dimensionLevel> !CL "," !
       CL ![ ^<resultMeasure> AS !] !CL "FROM" !E <base> !E !NJL <
       dimension> !NJL "WHERE" !AL ^<slice> !AL "GROUP BY" !CL <
       dimensionLevel> !CL"HAVING" !AL ^<having> !AL'.

olap:base a pl:PatternElement;
    rdfs:range qb:DataStructureDefinition, pl:RolapFactTable;
    pl:multiplicity pl:One.

olap:resultMeasure a pl:PatternElement;
    rdfs:range pl:CalculatedMeasureProperty, qb:MeasureProperty;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:slice a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:dimensionLevel a pl:PatternElement;
    rdfs:range qb4o:LevelProperty, pl:RolapDimensionLevel;
```

```
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:dimension a pl:PatternElement;
    rdfs:range qb:DimensionProperty, qbgen:ComplexAttribute, pl:
       RolapDimensionTable, pl:JoinElement;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:having a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.


## Set-Base-Comparison

olap:SetBaseComparison a pl:Pattern;
    pl:result [ pl:element olap:measure; pl:elementPrefix "Base_
       "];
    pl:result [ pl:element olap:measure; pl:elementPrefix "SI_"];
    pl:result olap:dimensionLevel;
    pl:hasElement olap:base, olap:baseSlice, olap:dimensionLevel,
        olap:dimension, olap:measure, olap:siSlice;
    pl:patternExpression '"WITH base AS (SELECT *"  "FROM" !E <
       Base> !E "WHERE" !CL <BaseSlice> !CL ")""SELECT " !CL <
       dimensionLevel> !CL "," !CL "Base_"!+<Measure> !CL "," !CL
       "SI_"!+ <Measure> !CL"FROM (SELECT" !CL <dimensionLevel> !
       CL "," !CL ![ ^<Measure> AS "Base_"!+ !] !CL "FROM base" !
       NJL <dimension> !NJL "GROUP BY" !CL <dimensionLevel> !CL ")
       " "NATURAL JOIN" "(SELECT" !CL <dimensionLevel> !CL "," !CL
        ![ ^<Measure> AS "SI_"!+!] !CL "FROM base" !NJL <dimension
       > !NJL "WHERE" !AL ^<SIslice> !AL "GROUP BY" !CL <
       dimensionLevel> !CL ")"'.

olap:baseSlice a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:measure a pl:PatternElement;
    rdfs:range pl:CalculatedMeasureProperty, pl:RolapMeasure;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:siSlice a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.
```

```
## Set-SuperSet-Comparison

olap:SetSuperSetComparsion a pl:Pattern;
    pl:result [ pl:element olap:siMeasure; pl:elementPrefix "SI_
       "];
    pl:result [ pl:element olap:scMeasure; pl:elementPrefix "SC_
       "];
    pl:result olap:siDimensionLevel;
    pl:hasElement olap:base, olap:baseSlice, olap:siMeasure, olap
       :siSlice, olap:siDimensionLevel, olap:siDimension,
        olap:scMeasure, olap:scSlice,   olap:scDimensionLevel,
           olap:scDimension, olap:compHaving;
    pl:patternExpression '"WITH base AS ("  "SELECT *"  "FROM" !E
        <Base> !E "WHERE" !CL <BaseSlice> !CL ")""SELECT " !CL <
       siDimensionLevel> !CL "," !CL "SI_"!+<siMeasure> !CL "," !
       CL "SC_"!+<scMeasure> !CL"FROM (SELECT" !CL <
       siDimensionLevel> !CL "," !CL ![ ^<siMeasure> AS "SI_"!+ !]
        !CL "FROM base" !NJL <siDimension> !NJL "WHERE" !AL ^<
       siSlice> !AL "GROUP BY"!CL <siDimensionLevel> !CL ")" "
       NATURAL JOIN" "(SELECT" !CL <scDimensionLevel> !CL "," !CL
       ![^<scMeasure> AS "SC_"!+!] !CL "FROM base" !NJL <
       scDimension> !NJL "WHERE" !AL ^<scSlice> !AL "GROUP BY" !CL
        <scDimensionLevel> !CL ")""WHERE" !CL ^<compHaving> !CL'.

olap:siMeasure a pl:PatternElement;
    rdfs:range pl:CalculatedMeasureProperty, qb:MeasureProperty;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:siDimensionLevel a pl:PatternElement;
    rdfs:range qb4o:LevelProperty, qbgen:ComplexAttribute;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:siDimension a pl:PatternElement;
    rdfs:range qb:DimensionProperty, qbgen:ComplexAttribute;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:scMeasure a pl:PatternElement;
    rdfs:range pl:CalculatedMeasureProperty, qb:MeasureProperty;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:scDimensionLevel a pl:PatternElement;
    rdfs:range qb4o:LevelProperty, qbgen:ComplexAttribute;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.
```

```
olap:scDimension a pl:PatternElement;
    rdfs:range qb:DimensionProperty, qbgen:ComplexAttribute;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:scSlice a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

olap:compHaving a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

## SetSetComparison

olap:SetSetComparison a pl:Pattern;
    pl:result [ pl:element olap:measure; pl:elementPrefix "SC_"];
    pl:result [ pl:element olap:measure; pl:elementPrefix "SI_"];
    pl:result olap:compMeasure;
    pl:result olap:dimensionLevel;
    pl:hasElement olap:base, olap:baseSlice, olap:measure, olap:
       dimensionLevel, olap:dimension,
    olap:siSlice, olap:scSlice, olap:compMeasure, olap:compHaving
       ;
    pl:patternExpression '"WITH base AS (   SELECT *    FROM" !E
       <base> !E  "WHERE" !CL ^<baseSlice> !CL ")""SELECT" !CL <
       dimensionLevel> !CL "," !CL "SI_"!+<Measure> !CL "," !CL "
       SC_"!+<Measure> !CL ","  !CL ![ ^<compMeasure> AS !] !CL"
       FROM (SELECT" !CL <dimensionLevel> !CL "," !CL ![ ^<Measure
       > AS "SI_"!+ !] !CL "FROM" !E <base> !E !NJL <dimension> !
       NJL "WHERE" !AL ^<SIslice> !AL "GROUP BY" !CL <
       dimensionLevel> !CL ")" "NATURAL JOIN" "(SELECT" !CL <
       dimensionLevel>!CL "," !CL ![ ^<Measure> AS "SC_"!+ !] !CL
       "FROM base" !NJL <dimension> !NJL " WHERE" !AL ^<SCslice> !
       AL "GROUP BY" !CL <dimensionLevel> !CL")""WHERE" !CL <
       compHaving> !CL'.

olap:compMeasure a pl:PatternElement;
    rdfs:range pl:CalculatedMeasureProperty, qb:MeasureProperty;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:base.

## Heterogenous Comprison

olap:HeterogenousComparison a pl:Pattern;
```

```
    pl:result [ pl:element olap:siMeasure; pl:elementPrefix "SI_
       "];
    pl:result [ pl:element olap:scMeasure; pl:elementPrefix "SC_
       "];
    pl:result olap:hSiDimensionLevel;
    pl:hasElement olap:hSiBase, olap:hSiMeasure, olap:
       hSiDimensionLevel, olap:hSiDimension,
    olap:hSiSlice, olap:hScBase, olap:hScMeasure, olap:
       hScDimensionLevel, olap:hScDimension,
    olap:hScSlice, olap:hFactCorrelation;
    pl:patternExpression '"SELECT" !CL ![ "SI."!+<
       hSiDimensionLevel> AS !] !CL "," !CL ![ "SI."!+<hSiMeasure>
        AS "SI_"!+ !] !CL "," !CL ![ "SC."!+<hScMeasure> AS "SC_
       "!+ !] !CL "FROM (SELECT" !CL <hSiDimensionLevel> !CL "," !
       CL ![ ^<hSiMeasure> AS !] !CL "FROM" !E <siBase> !E !NJL <
       hSiDimension> !NJL "WHERE" !AL <hSiSlice> !AL "GROUP BY" !
       CL <hSiDimensionLevel> !CL ") SI, (SELECT" !CL <
       hScDimensionLevel> !CL "," !CL ![ ^<hScMeasure> AS !] !CL "
       FROM" !E <scBase> !E !NJL <hScDimension> !NJL "WHERE" !AL
       ^<hScSlice> !AL "GROUP BY" !CL <hScDimensionLevel> !CL ")
       SC WHERE" !AL ^<factCorrelation> !AL'.


olap:hScBase a pl:PatternElement;
    rdfs:range qb:DataStructureDefinition;
    pl:multiplicity pl:One.

olap:hSiBase a pl:PatternElement;
    rdfs:range qb:DataStructureDefinition;
    pl:multiplicity pl:One.

olap:hSiMeasure a pl:PatternElement;
    rdfs:range pl:CalculatedMeasureProperty, qb:MeasureProperty;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:hSiBase.

olap:hScMeasure a pl:PatternElement;
    rdfs:range pl:CalculatedMeasureProperty, qb:MeasureProperty;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:hScBase.

olap:hScDimensionLevel a pl:PatternElement;
    rdfs:range qb4o:LevelProperty, qbgen:ComplexAttribute;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:hScBase.

olap:hScDimension a pl:PatternElement;
    rdfs:range qb:DimensionProperty, qbgen:ComplexAttribute;
    pl:multiplicity pl:OneOrMore;
```

```
    pl:partOf olap:hScBase.

olap:hSiDimensionLevel a pl:PatternElement;
    rdfs:range qb4o:LevelProperty, qbgen:ComplexAttribute;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:hSiBase.

olap:hSiDimension a pl:PatternElement;
    rdfs:range qb:DimensionProperty, qbgen:ComplexAttribute;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:hSiBase.

olap:hScSlice a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:hScBase.

olap:hSiSlice a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:hSiBase.

olap:hFactCorrelation a pl:PatternElement;
    rdfs:range pl:Predicate;
    pl:multiplicity pl:OneOrMore;
    pl:partOf olap:hSiBase;
    pl:partOf olap:hScBase.

############################
## PatternInstances
############################

olap:Farmsite_1306707 a pl:Predicate;
    pl:expression "FarmSiteId='1306707'";
    pl:uses agri:FarmSite.

olap:AVG_over_15 a  pl:Predicate;
    pl:expression "AVG(Milkyield_Parlour)>15";
    pl:uses agri:Milkyield_Parlour.

olap:Avg_Milkyield_ROLAP a  pl:RolapPatternInstance;
    pl:instanceOf olap:NonComparative;
    pl:persistAs pl:Snapshot;
    olap:resultMeasure
        olap:Avg_Milkyield;
    olap:base "Milk";
    olap:slice olap:Farmsite_1306707;
    olap:dimensionLevel
        "FarmSiteId", "NationalId";
```

```
    olap:dimension
        [pl:table "Farmsite"; pl:order 1; pl:join "LEFT"],  "
          Animal";
    olap:having olap:AVG_over_15.

olap:calvingNo a  pl:CalculatedMeasureProperty;
    pl:expression "MAX(CalvingNo)".

## Non-Comparative Query
## The average dayly milkyield per Animal of a specific farm

olap:Avg_Milkyield a pl:CalculatedMeasureProperty;
    pl:expression "AVG(Milkyield_Parlour)";
    pl:uses agri:Milkyield_Parlour.

olap:Farmsite_1 a pl:Predicate;
    pl:expression "FarmSiteId='12345'";
    pl:uses agri:FarmSite.

olap:AVG_over_40 a  pl:Predicate;
    pl:expression "AVG(Milkyield_Parlour)>40";
    pl:uses agri:Milkyield_Parlour.

olap:EmptyCondition a pl:Predicate;
    pl:expression "1=1".


olap:Avg_Milyield_QB a pl:QbPatternInstance;
    pl:instanceOf olap:NonComparative;
    pl:persistAs pl:View;
    olap:resultMeasure olap:Avg_Milkyield;
    olap:base agri:Milk;
    olap:slice olap:Farmsite_1306707;
    olap:dimensionLevel agri:Animal, agri:Date_, agri:FarmSite,
       agri:MainBreed;
    olap:dimension agri:AnimalDim, agri:Date_Dim, agri:
       FarmSiteDim, agri:Enterprise;
    olap:having olap:AVG_over_15.

## Non-Comparative Query
## The average

olap:Avg_Weight a pl:CalculatedMeasureProperty;
    pl:expression "AVG(CalfWeight)";
    pl:uses agri:CalfWeight.

olap:Avg_CalvingWeight a pl:QbPatternInstance;
    pl:instanceOf olap:NonComparative;
    olap:resultMeasure olap:Avg_Weight;
```

```
    olap:base agri:Calving;
    olap:slice olap:Farmsite_1;
    olap:dimensionLevel agri:Animal;
    olap:dimensionLevel agri:Calf;
    olap:dimensionLevel agri:Date_;
    olap:dimension [pl:dimension agri:AnimalDim; pl:inFact agri:
       Calving_CalfWeight];
    olap:dimension [pl:dimension agri:CalfDim; pl:inFact agri:
       Calving_CalfWeight];
    olap:dimension agri:Date_;
    olap:having olap:EmptyCondition.


## (COUNT all Animals and Count Animals with HIGHFAT) per
   Farmsite On date=X
##SET-BASE-Comparison

olap:Animal_Count a pl:CalculatedMeasureProperty;
    pl:expression "COUNT(NationalId)";
    pl:uses "NationalId".

olap:HighFat_DHIA a pl:Predicate;
    pl:expression "FatContent_DHIA > 5";
    pl:uses "FatContent_DHIA".

olap:LactationLastMonth a pl:Predicate;
    pl:expression "DayOfLactation < 30";
    pl:uses "DayOfLactation".

olap:DHIA_Fat_CT_under_30 a pl:Predicate;
    pl:expression "Milk_DHIA_FatContent_CT < 30";
    pl:uses "Milk_DHIA_FatContent_CT".

olap:20160824 a pl:Predicate;
    pl:expression "Date_='24.08.2016'";
    pl:uses "Date_".


olap:Comparison_HighFat a pl:RolapPatternInstance;
    pl:instanceOf olap:SetBaseComparison;
    olap:base "Milk";
    olap:baseSlice olap:20160824;
    olap:dimensionLevel "FarmSiteId";
    olap:dimension "FarmSite";
    olap:measure olap:Animal_Count;
    olap:siSlice olap:HighFat_DHIA;
    olap:siSlice olap:LactationLastMonth;
    olap:siSlice olap:DHIA_Fat_CT_under_30.
```

```
olap:Comp_HighFat a pl:QbPatternInstance;
    pl:instanceOf olap:SetBaseComparison;
    olap:base agri:Milk;
    olap:baseSlice olap:20160824;
    olap:dimensionLevel agri:FarmSite;
    olap:dimension agri:FarmSiteDim;
    olap:measure olap:Animal_Count;
    olap:siSlice olap:HighFat_DHIA;
    olap:siSlice olap:LactationLastMonth;
    olap:siSlice olap:DHIA_Fat_CT_under_30.s



## Milyield of Animal compared to STDDEV_Milkyield of FarmSite
## SET-SUPERSET-COMPARISON

olap:StdDev_Milkyield a pl:CalculatedMeasureProperty;
    pl:expression "STDDEV(Milkyield_Parlour)";
    pl:uses agri:Milkyield_Parlour.

olap:Max_Milkyield a pl:CalculatedMeasureProperty;
    pl:expression "MAX(Milkyield_Parlour)";
    pl:uses agri:Milkyield_Parlour.

olap:20160717 a pl:Predicate;
    pl:expression "Date_='17.04.16'";
    pl:uses agri:Date_.

olap:compMilkyield_to_STDEV a pl:Predicate;
    pl:expression "SI_Max_Milkyield < 2*SC_STDDEV_Milkyield";
    pl:uses olap:Max_Milkyield, olap:StdDev_Milkyield.

olap:STDEV_milkyield_ROLAP a pl:ROLAPPatternInstance;
    pl:instanceOf olap:SetSuperSetComparsion;
    olap:base "Milk_Parlour_Milkyield";
    olap:baseSlice olap:20160717;
    olap:baseSlice olap:LactationLastMonth;
    olap:siMeasure olap:Max_Milkyield;
    olap:siSlice olap:EmptyCondition;
    olap:siDimensionLevel "NationalId","FarmSiteId";
    olap:siDimension "Animal", "FarmSite";
    olap:scMeasure olap:StdDev_Milkyield;
    olap:scSlice olap:EmptyCondition;
    olap:scDimensionLevel "FarmSiteId";
    olap:scDimension "FarmSite";
    olap:compHaving olap:compMilkyield_to_STDEV.

olap:STDEV_milkyield_QB a pl:QbPatternInstance;
```

```
    pl:instanceOf olap:SetSuperSetComparsion;
    olap:base agri:Milk_Parlour_Milkyield;
    olap:baseSlice olap:20160717;
    olap:baseSlice olap:LactationLastMonth;
    olap:siMeasure olap:Max_Milkyield;
    olap:siSlice olap:EmptyCondition;
    olap:siDimensionLevel agri:Animal, agri:FarmSite;
    olap:siDimension agri:AnimalDim, agri:FarmSiteDim;
    olap:scMeasure olap:StdDev_Milkyield;
    olap:scSlice olap:EmptyCondition;
    olap:scDimensionLevel agri:FarmSite;
    olap:scDimension agri:FarmSiteDim;
    olap:compHaving olap:compMilkyield_to_STDEV.

## Delta-BCS of today and 30 days ago
##SET-SET-COMPARISON

olap:Max_BCS a pl:CalculatedMeasureProperty;
    pl:expression "MAX(BCS)";
    pl:uses agri:BCS.

olap:Delta_BCS a  pl:CalculatedMeasureProperty;
    pl:expression "SI_MAX_BCS-SC_MAX_BCS";
    pl:uses olap:Max_BCS.

olap:DatePeriod a pl:Predicate;
    pl:expression "Date_ BETWEEN '1.1.2016' AND '1.1.2017'";
    pl:uses agri:Date_.

olap:DayOfLactation_no1 a pl:Predicate;
    pl:expression "DayOfLactation=1";
    pl:uses agri:DayOfLactation.

olap:DayOfLactation_no30 a pl:Predicate;
    pl:expression "DayOfLactation=30";
    pl:uses agri:DayOfLactation.

olap:DELTA_BCS_All_Animals a pl:QbPatternInstance;
    pl:instanceOf olap:SetSetComparison;
    olap:base agri:BodyCondition;
    olap:baseSlice olap:DatePeriod;
    olap:measure olap:Max_BCS;
    olap:dimensionLevel agri:Animal, agri:CalvingNo;
    olap:dimension agri:AnimalDim, agri:CalvingNoDim;
    olap:siSlice olap:DayOfLactation_no1;
    olap:scSlice olap:DayOfLactation_no30;
    olap:compMeasure olap:Delta_BCS;
    olap:compHaving olap:EmptyCondition.
```

```
##Heteorgenouse Base Cubes

##Correlation to Offer of Roughage and Milkyield of the next day

olap:Avg_Amount_Offer_Rou a pl:CalculatedMeasureProperty;
    pl:expression "MAX(Amount_Offer_Approx_Rou)";
    pl:uses agri:Amount_Offer_Approx_Rou.

olap:MatchTodayTommorrow a pl:Predicate;
    pl:expression "SI.Date_ = SC.Date_+1";
    pl:uses agri:Date_.

olap:Food_Milkyield_nextDay a pl:QbPatternInstance;
    pl:instanceOf olap:HeterogenousComparison;
    olap:hSiBase agri:Milk;
    olap:hSiMeasure olap:Avg_Milkyield;
    olap:hSiDimension agri:Date_Dim,agri:AnimalDim, agri:
       FarmSiteDim;
    olap:hSiDimensionLevel agri:Date_,agri:Animal, agri:FarmSite;
    olap:hSiSlice olap:EmptyCondition;
    olap:hScBase agri:Feeding;
    olap:hScMeasure olap:Avg_Amount_Offer_Rou;
    olap:hScDimension agri:Date_Dim,agri:AnimalDim, agri:
       FarmSiteDim;
    olap:hScDimensionLevel agri:Date_,agri:Animal, agri:FarmSite;
    olap:hScSlice olap:EmptyCondition;
    olap:hFactCorrelation olap:MatchTodayTommorrow.

## Amount of Animals per FarmSite which were diagnosed
## with Ketosis in the 30 days after Calving

olap:Ketosis a pl:Predicate;
    pl:expression "EventName='Ketosis'";
    pl:uses agri:Event.

olap:MatchDateIn30AnimalFarmsite a pl:Predicate;
    pl:expression "SI.Date_ <= SC.Date_+30 AND SI.NationalId=SC.
       NationalId AND SI.FarmSiteId=SC.FarmSiteId";
    pl:uses agri:Date_,agri:FarmSite, agri:Animal.

olap:Food_Milkyield_nextDay a pl:QbPatternInstance;
    pl:instanceOf olap:HeterogenousComparison;
    olap:hSiBase agri:Calving;
    olap:hSiMeasure olap:Animal_Count;
    olap:hSiDimension agri:Date_Dim, agri:FarmSiteDim;
    olap:hSiDimensionLevel agri:Date_, agri:FarmSite;
    olap:hSiSlice olap:EmptyCondition;
    olap:hScBase agri:Occurred;
    olap:hScMeasure olap:Animal_Count;
```

```
olap:hScDimension agri:Date_Dim, agri:FarmSiteDim;
olap:hScDimensionLevel agri:Date_, agri:FarmSite;
olap:hScSlice olap:Ketosis;
olap:hFactCorrelation olap:MatchDateIn30AnimalFarmsite.
```