JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

SOWI
Sozial- und Wirtschaftswissenschaftliche
Fakultät

An OLAP API for Cubes with Ontology-Valued Measures

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Masterstudium

WIRTSCHAFTSINFORMATIK

Eingereicht von:

Michael Schnepf

Angefertigt am:

Institut für Wirtschaftsinformatik – Data & Knowledge Engineering

Beurteilung:

o.Univ.-Prof. Dr. Michael Schrefl

Mitbetreuung, Mitwirkung:

Dr. Christoph Schütz

Linz, August 2015

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, August 2015 Michael Schnepf

Abstract. Traditional OLAP systems operate on numeric values only. Many scenarios, however, are represented more appropriately by the use of business model ontologies. The extension of OLAP systems to support operations on business model ontologies offers new possibilities in data analytics. We use ontology-valued measures to support the analysis of non-numeric values. An API is created in order to execute different OLAP operators on ontology-valued measures.

Kurzfassung. Die Operatoren traditioneller OLAP-Systeme unterstützen lediglich numerische Werte. Manche Daten können jedoch besser durch Business Model Ontologien dargestellt werden. Eine Erweiterung von OLAP-Systemen hinsichtlich der Unterstützung von Business Model Ontologien bietet neue Möglichkeiten für die Datenanalyse. Wir verwenden Ontology-Valued Measures um die Analyse von nicht-numerischen Daten zu ermöglichen. Eine API wird entwickelt um unterschiedliche OLAP-Operatoren auf Ontology-Valued Measures ausführen zu können.

Content

Figures

Listings

Tables

1 Introduction

Online Analytical Processing (OLAP) systems are used to analyse data in order to generate new insights into business. Those systems are able to process data in a way to support companies in their decision making process. They enable an analyst to view data within different dimensions and levels and to perform operations on the base data. The functionality of the system is based on a multidimensional model. This model stores the dimensional information in a hierarchically structured way to analyse base data within different dimensions and levels. The combination of the base data and the dimensional information builds the OLAP cube. Figure 1 shows an example OLAP cube. The OLAP cube consists of different dimensions De*partment*, *Location* and *Time*. For every dimension there exist different attributes on specific levels which roll up hierarchically, e.g. Linz (*City)*, Austria (*Country)* and Europe (*Continent)* for the dimension *Location*. For example, analysts are able to analyse the *costs* of the department *Production*, located in *Austria* in the year *2013*.

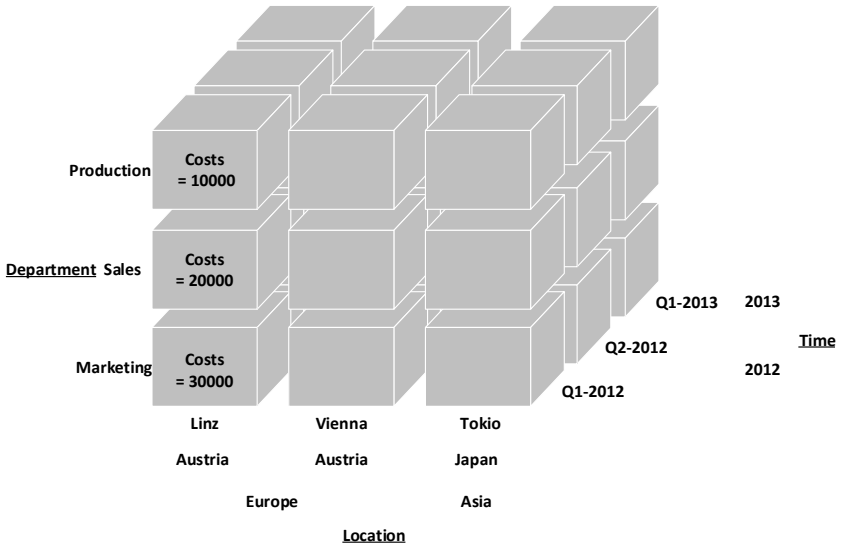

Figure 1: OLAP cube

Conventional OLAP systems operate on numeric values only. Many business scenarios cannot be expressed by numeric values. The power of OLAP systems would increase if they operated on information not only represented by numeric measures. Those information may be formalized by business model ontologies in order to represent business scenarios. To enable OLAP operations on business model ontologies the OLAP cubes need to be extended with ontology-valued measures [1].

We develop an application programming interface (API) to enable OLAP operations on business models. The remainder of this thesis is structured as follows. Chapter 2 covers the basic concepts used in this thesis and gives an overview about related work. In Chapter 3 the basic concepts are illustrated which enable multidimensional modelling with OLAP cubes. In Chapter 4 the developed operators are demonstrated by the use of different example scenarios. In Chapter 5 the technical implementation of the API and the operators are illustrated. Chapter 6 summarizes the main points of the thesis and provides an outlook on future work.

2 Background

This chapter covers essential concepts which have been used to implement the API. The different sections focus only on those aspects of the concepts which are relevant and do not explain every single detail of the concepts. Specific details of the concepts will be explained in respective sections. Furthermore, this chapter outlines related work in order to emphasize the relevance of this thesis.

2.1 RDF(S) and SPARQL

The Resource Description Framework (RDF) [2] is used to define information of the World Wide Web (WWW) in a standardized form. Therefore, it uses triples which consist of a subject and an object connected by a predicate, also called property. The elements of a triple are also called resources. Those resources may be of type Internationalized Resource Identifier (IRI) whereas the object also can be a literal. Subjects and objects may also be blank nodes. To enhance readability the IRI may be shortened by the use of namespace prefixes which are defined for an IRI. For example, the IRI *http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral* may be shortened by using the namespace prefix *rdf:* which leads to the shortened use *rdf:XMLLiteral*.

For RDF resources it is possible to define a data modelling vocabulary with RDF Schema (RDFS) [3]. With RDFS it is possible to define structure and relationships of resources by using RDF terms. The basic RDFS concepts are classes and properties. Classes may be used to group specific resources. Properties are used to define relationships between classes or resources. For example, the property *subClassOf* defines inheritance between RDFS classes, the properties *domain* and *range* define the type of classes a property is able to connect, respectively.

SPARQL 1.1 [4] provides languages and protocols to query and update RDF data. A SPARQL select query selects tuples and binds them to variables which represent the result of the query. The result can be printed or may be used to update existing RDF resources by SPARQL update statements. Another form of SPARQL queries are ASK queries. Their result is a Boolean value which indicates whether a query pattern exists or not.

2.2 Contextualized Knowledge Repository

In the WWW there exist big amounts of knowledge represented as RDF datasets hosted by services like DBpedia[1]. An RDF dataset consists of different graphs. There exists exactly one default graph and an arbitrary number of named graphs. The default graph has no name whereas named graphs are specified by an IRI. With named graphs, knowledge may be separated and it is possible to express meta-information about the graphs [5]. Contextualized Knowledge Repositories (CKR) [6] use named graphs and add meta-information to define the context within which the knowledge is valid. Assume that there exists knowledge about all world cup champions. The term world cup champion is well defined but it cannot be clearly determined without the use of context, e.g. year and type of sports. For example, Germany (2014) and Spain (2010) both won the soccer wold cup, whereas Poland (2014) and Italy (2010) won the volleyball world cup. Without the use of contextual information it is not possible to determine the soccer world cup champion of 2010.

To add contextual information to the knowledge the *context as a box* paradigm [6] is used. This paradigm differentiates between original knowledge located inside the *box* and contextual knowledge located outside the *box*. The contextual knowledge defines the context in which the original knowledge is valid. Referring to the example above, the original knowledge inside the box is the definition of the countries as world cup champions. The contextual knowledge located outside the box are information about the year and the type of sport.

In this thesis we use named graphs in order to separate knowledge and to add contextual information to them. Furthermore the CKR framework[2], in combination with a custom ruleset provided by FBK[3], is used to generate additional knowledge about the contextual information and the relations between the graphs.

2.3 Business Model Ontologies

In order to enable OLAP systems operating on business scenarios the knowledge of the business scenarios need to be formalized. Business model ontologies “capture the complex interdependencies between business objects” [1, p. 514]. They describe the elements and their relationships which are consumed or produced by a company in

[1] http://wiki.dbpedia.org/
[2] https://dkm.fbk.eu/technologies/ckr
[3] http://www.fbk.eu/

order to “generate profitable and sustainable revenue streams” [7, p. 15]. For modelling business scenarios the REA business model ontology [8] may be used. It focuses on **R**esources, **E**vents and **A**gents. Internal agents provide resources to external agents in order to receive resources with a higher value than the one provided. The exchange of a resource is called event and has to occur in duality with another event. The example models in this thesis are inspired by the REA ontology. Note, however, that a strict representation according to the REA ontology is not intended.

2.4 Related Work

The analysis of business scenarios not boiling down to numeric measures requires the adaption of traditional OLAP systems. By applying semantic technologies, business analysts are able to gain new insights into business. The Semantic Cockpit project [9] uses the DFM for multidimensional modelling. By using reasoning capabilities it is possible to formulate OLAP queries and to interpret the results for further analytics. The API for cubes with ontology-valued measures focuses on the execution of OLAP operators. They offer several possibilities in configuration to define the desired level of abstraction based on the multidimensional model. The operators implemented by the API are similar to aggregated RDF views [10]. Roll up operators need to be configured with dimensional information to return views containing aggregated resources.

Graph OLAP [11] enables the analysis of graphs within different perspectives. It combines graphs with multidimensional information. The supported informational roll up and topological roll up are similar to the merge and abstract operators of our introduced API. Graph OLAP uses weighted graphs which are not intended for the representation of complex business scenarios [1]. In OLAP systems with ontology-valued measures, business model ontologies represent business scenarios.

Abelló et al. [12] identify challenges for the enrichment of traditional OLAP systems with semantic web data and the use of semantic technologies for data integration for OLAP systems. This leads to Exploratory OLAP systems which should be able to receive and process different kinds of (semi-) structured data, to combine the data with multidimensional information and to query using OLAP dimensions. The API presented in this thesis focuses on the execution of OLAP operators on RDF data, implemented using SPARQL queries. Such an OLAP system assumes the existence of already integrated knowledge. The issue of data integration for OLAP systems with ontology-valued measures will be tackled by future work.

3 OLAP Cubes with Ontology-Valued Measures

Conventional OLAP systems need to be adapted in order to operate on non-numeric measures. For example, a company may be single manufacturer of a specific product in Austria. To maximize profit the company decides to additionally export the products to all countries of Europe. While there is no competing company in Austria this may not be true for the other countries. The company needs to include knowledge from all European countries in order to find potential competitors.

3.1 Base Facts and Shared Facts

In order to analyse base data within different dimensions a dimensional model needs to be defined. Similar to Schütz et al. [1] the Dimensional Fact Model (DFM) [13] is used. The example model shown in Figure 2 consists of the fact schema *Sales* which represents a specific business scenario formalized by a business model ontology. Furthermore, there are three different dimensions *Location*, *Department* and *Time*. Those dimensions define the context in which the fact is valid. Every dimension is described by levels which roll up to each other and are ordered in a hierarchical way from most to least granular. The directed arrows between the levels represent possible ways of aggregation, e.g. the *Sales* fact may be aggregated on the levels *Continent*, *Department* and *Year*.

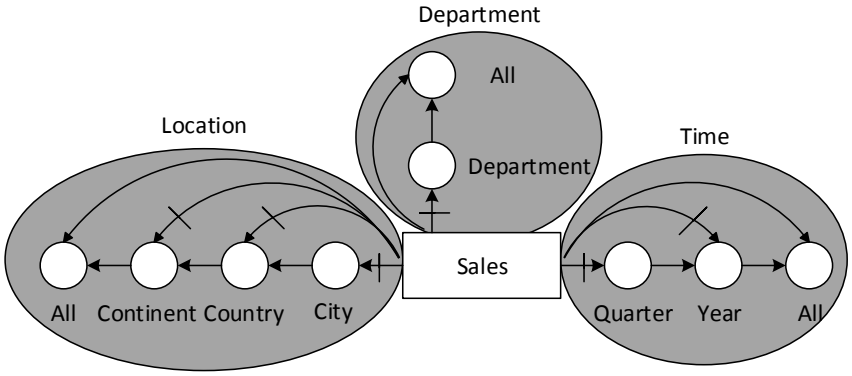

Figure 2: DFM representation of the Dimensional Model

It is necessary to differentiate between different kinds of facts. Facts at finest level of granularity (*base facts*) and facts at coarser level of granularity (*shared facts*). Due to the hierarchical organization of the dimension attributes and the levels in the DFM there is also a hierarchical organization of facts. So all information of *FactA* with the dimension *Location* and the dimension attribute *Europe* is also valid for *FactB* with the dimension *Location* and the dimension attribute *Linz*. So *FactB* inherits all information from *FactA* which means that facts at higher levels of granularity may be called *shared facts*.

The DFM defines that the base data, also called instances of the fact schema, need to be present at the most granular level. So it is mandatory that the instance of the fact schema *Sales* is defined at level *City*, *Department* and *Quarter*. We use multigranular cubes similar to multilevel cubes [14] that enable facts at multiple levels of abstraction. Therefore optional aggregation paths are used which are shown by the directed arrows marked with a dash. Due to that enhancement of the DFM model it is now possible that an instance of the fact schema *Sales* belongs to a whole *Continent* rather than to a specific *City*.

3.2 Multidimensional Modelling

In order to develop a multidimensional model in a way it can be understood and interpreted by a machine it needs to be formalized. Therefore we translate the model from Figure 2 into an RDF representation based on the representation used for the Semantic Cockpit Project [15].

3.2.1 Basic OLAP vocabulary

The first step of defining a multidimensional model is to formulate the basic OLAP vocabulary. With this vocabulary it is possible to represent the multidimensional model of Figure 2. Therefore facts need to be defined, quantified by dimension attribute values at specific levels. The hierarchical organization of both the dimension attribute values and the levels as well as the assertion of modules to contexts need to be part of the vocabulary. Listing 1 shows an example vocabulary. *DimensionAttributeValue*, *Level* and *Fact* (Line 1-6) are the main classes which need to be defined and correspond to the concepts already mentioned in Section 3.1. Due to the fact that *DimensionAttributeValues* and *Levels* are different kind of resources, this needs to be declared explicitly. This is defined by the statement *disjointWith Level* (Line 4) and means that a resource which is a *DimensionAttributeValue* cannot be a *Level.* The

```
1  :DimensionAttributeValue
2                        rdf:type              owl:Class ;
3                        rdfs:subClassOf       ckr:AttributeValue;
4                        owl:disjointWith      :Level .
5  :Level                rdf:type              owl:Class ;
6  :Fact                 rdfs:subClassOf       ckr:Context .
7
8  :rollsUpTo            rdf:type              owl:ObjectProperty .
9  :directlyRollsUpTo    rdf:type              owl:ObjectProperty ;
10                       rdfs:subPropertyOf    :rollsUpTo .
11 :hasAssertedModule    rdfs:subPropertyOf    ckr:hasModule .
12 :hasDimensionAttributeValue
13                       rdf:type              owl:ObjectProperty ;
14                       rdfs:range     :DimensionAttributeValue ;
15                       rdfs:domain           :Fact .
16 :atLevel       rdf:type                     owl:ObjectProperty ;
17                       rdfs:domain    :DimensionAttributeValue ;
18                       rdfs:range            :Level .
```

Listing 1: Basic OLAP vocabulary

hierarchical organization of both the *Levels* and the *DimensionAttributeValues* is defined by the property *directlyRollsUpTo*, a sub property of *rollsUpTo* (Lines 8-10). In order to assert *Modules* to *Facts*, the *hasAssertedModule* property is used which is a sub property of *hasModule* (Line 11). *DimensionAttributeValues* are asserted to *Facts* by the use of the property *hasDimensionAttributeValue* (Lines 12-15). The property *atLevel* (Lines 16-18) defines the *Level* of a *DimensionAttributeValue*. For example, the *Level* of *Linz* is *City* and the *Level* of *Europe* is *Continent*.

Listing 2 illustrates the definition of dimensions and properties to assign dimension attribute values to contexts. Lines 1-9 show an example for the definition of the dimensions *Department*, *Time* and *Location*. Those classes are defined as subclass of *DimensionAttributeValue*. To prevent wrong assignment of individuals to the appropriate classes, the *disjointWith* property prevents that the assertion of same individuals to different classes. To assign *DimensionAttributeValues* to *Facts* the properties *hasDepartment, hasTime* and *hasLocation* (Lines 10-21) are used. The properties are defined as subproperty of *hasDimensionAttributeValue.* For the *hasDimensionAttributeValue* property the domain is defined as *Fact.* Because the properties *hasDepartment, hasTime* and *hasLocation* define the range as the dimensions *Department*, *Time* and *Location,* respectively, a *Fact* can thus be assigned to its specific dimension attribute values. The statement *hasDepartment type FunctionalProperty* defines that the property *hasDepartment* needs to have a unique value for each instance. So it is impossible to define different dimension attributes for the same fact.

```
1   :Department       rdf:type              owl:Class ;
2                     owl:disjointWith      :Time, :Location ;
3                     rdfs:subClassOf       :DimensionAttributeValue .
4   :Time             rdf:type              owl:Class ;
5                     owl:disjointWith      :Location ;
6                     rdfs:subClassOf       :DimensionAttributeValue .
7
8   :Location         rdf:type              owl:Class ;
9                     rdfs:subClassOf       :DimensionAttributeValue.
10  :hasDepartment    rdf:type              owl:FunctionalProperty ,
11                                          owl:ObjectProperty ;
12                    rdfs:range            :Department ;
13                    rdfs:subPropertyOf    :hasDimensionAttributeValue.
14  :hasTime          rdf:type              owl:FunctionalProperty ,
15                                          owl:ObjectProperty ;
16                    rdfs:range            :Time ;
17                    rdfs:subPropertyOf    :hasDimensionAttributeValue.
18  :hasLocation      rdf:type              owl:FunctionalProperty,
19                                          owl:ObjectProperty ;
20                    rdfs:range            :Location ;
21                    rdfs:subPropertyOf    :hasDimensionAttributeValue.
```

Listing 2: Example for OLAP dimensions

Now specific *Levels* and *DimensionAttributeValues* need to be defined. This is done by the use of *NamedIndividuals* which roll up to each other in order to represent the hierarchical organization of both *Levels* and *DimensionAttributeValues*. Listing 3 shows an example of the *DimensionAttributeValues* and *Levels*. Levels are *NamedIndividuals* of type *Level* (Line 1-10) which define the level of the dimension attribute. The property *directlyRollsUpTo* defines the hierarchical organization of the levels, e.g. *Level_Location_Continent directlyRollsUpTo Level_Location_All. NamedIndividuals* (Line 11-26) also need to define the *NamedIndividuals* they roll up to by the use of the property *directlyRollsUpTo*. Furthermore, the *Level* of the *DimensionAttributeValue* needs to be defined. For example, the *Level* of the *NamedIndividual Location_Europe* of type *Location* may be *Level_Location_Continent* and *Location_Europe* directly rolls up to *Location_All.* The Level of *Location_Austria* of type *Location* may be *Level_Location_City* and *Location_Austria* directly rolls up to *Location_Europe.*

```
:Level_Location_All       rdf:type      :Level ,
                                        owl:NamedIndividual .
:Level_Location_Continent rdf:type      :Level ,
                                        owl:NamedIndividual;
                          :directlyRollsUpTo
                                        :Level_Location_All .
:Level_Location_Country   rdf:type      :Level ,
                                        owl:NamedIndividual ;
                          :directlyRollsUpTo
                                        :Level_Location_Continent.
:Location_All             rdf:type      :Location ,
                                        owl:NamedIndividual ;
                          :atLevel
                                        :Level_Location_All .
:Location_Europe          rdf:type      :Location ,
                                        owl:NamedIndividual ;
                          :atLevel
                                        :Level_Location_Continent;
                          :directlyRollsUpTo
                                        :Location_All .
:Location_Austria         rdf:type      :Location ,
                                        owl:NamedIndividual ;
                          :atLevel
                                        :Level_Location_Country ;
                          :directlyRollsUpTo
                                        :Location_Europe.
```

Listing 3: Example of OLAP dimension attributes

3.2.2 Contexts and Modules

Note that a context is a specific coordinate in the OLAP cube which contains a fact. So facts are only valid in a specific context. It is not important to differentiate between facts and contexts so those terms will be used as synonyms further on.

Contexts are generated by the CKR framework. Therefore, the framework uses a custom ruleset provided by FBK. The ruleset is based on the materialization calculus [16] and is able to handle three dimensions Department, Location and Time. The ruleset needs to be extended in order to handle additional dimensions. The API, however, supports arbitrary amounts of dimensions.

For every combination of dimension attributes a specific context is generated. The context is represented by a named graph which contains the concatenated dimension attributes in its IRI. For example, based on the dimension attributes *Department_All, Location_Austria, Location_Europe, Location_All ,Time_Y2013* and *Time_All* the following contexts will be generated: *Ctx-Department_All-Location_Austria-Time_Y2013, Ctx-Department_All-Location_Austria-Time_All, Ctx-Department_All-*

Location_Europe-Time_Y2013, Ctx-Department_All-Location_Europe-Time_All, Ctx-Department_All-Location_All-Time_Y2013, Ctx-Department_All-Location_Europe-Time_All. Furthermore, those contexts are asserted with dimension attributes by the use of their respective properties to make them queryable.

Modules need to be defined explicitly in the dimensional model. This is done by defining a named graph for each module. The content are RDF(S) triples which describe a specific business scenario. To support the understandability of modules and contexts asserted, the dimension attributes are concatenated into the module name. Listing 4 shows an example of how a module may be defined.

```
1   :Module-Department_All-Location_All-Time_All
2   {
3    …
4   }
```

Listing 4: Example Module definition

We introduce a modeling guideline about blank nodes used by the API in order to represent RDF reification[4]. Blank nodes may be handled in a different way by different frameworks[5]. A problem arises if blank nodes with equal properties and objects are inserted into the same named graph. Some frameworks check the properties and objects of a blank node. If there already exists a blank node with equal properties and objects, no additional blank node is generated. Other frameworks simply generate new blank nodes independently of the properties and objects. This problem is solved by replacing bank nodes by generated IRIs as shown in Listing 5. If there exist more than one blank nodes with the same IRI, properties and objects, every framework treats them as same blank nodes.

```
1  :Module-Department_All-Location_All-Time_All
2  {
3    :blankNode1    rdf:subject :S;
4                   rdf:property :P;
5                   rdf:object :O;
6                   :count :C;
7  }
```

Listing 5: Example definition of blank nodes in modules

[4] http://www.w3.org/TR/rdf-schema/#ch_reificationvocab
[5] http://www.w3.org/TR/rdf11-concepts/#section-blank-nodes

The last step is to assign the generated context to the defined module as illustrated in Listing 6. This is done by the property *hasAssertedModule*. The property *hasModule* may not be used for the specific assertion of contexts to modules, because *hasModule* is used by the CKR framework. Due to the fact that it is necessary to distinguish between explicitly defined knowledge and knowledge generated by the CKR framework, two different properties need to be used. Listing 6 shows the assertion of a context to a module on the highest level of granularity.

```
1   :Ctx-Department_All-Location_All-Time_All
2                   :hasAssertedModule
3                   :Module-Department_All-Location_All-Time_All
```

Listing 6: Assertion of context and module

3.3 Online Analytical Processing

The OLAP API for cubes with ontology-valued measures [1] should provide operators which make it easy for a potential analyst to generate new insights into business scenarios. The operators align to traditional OLAP operators like slice, dice and roll up [17]. In this chapter the operators will be explained and differences to the traditional operators will be illustrated.

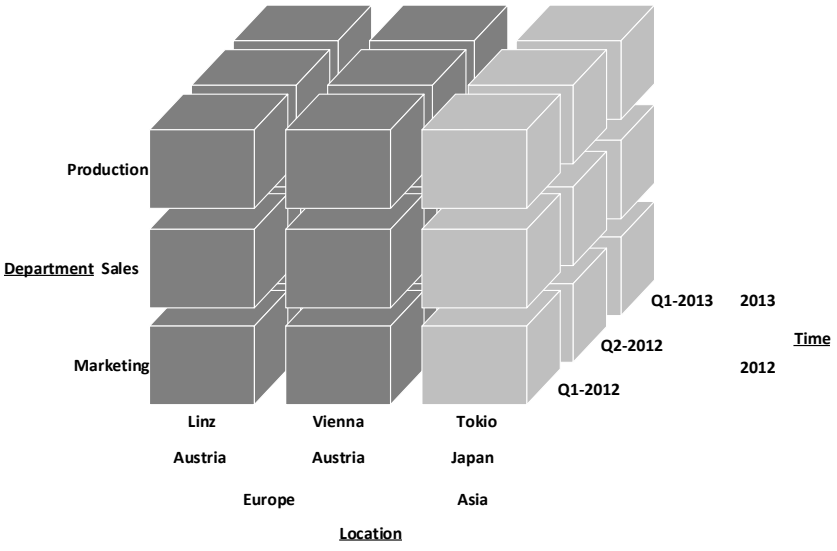

Figure 3: OLAP Slice operator

The first operator slice/dice is very similar to the slice and dice operation of traditional OLAP systems. The main idea of this operator is to select specific parts of the base data which are interesting for the analyst. Generally this is done by applying one or more dimensional attributes to the operator. If only a single dimensional attribute is applied the operator acts like the slice operator of traditional OLAP systems. Figure 3 shows the selected data if, for example, only the dimension attribute *Europe* is applied. If the resulting data should be additionally limited by the Year *2012*, all data which correspond to *2013* will not show up in the result. This is equal to the dice operation of traditional OLAP system where a specific cube of the base data is selected.

The merge operator is the first possibility to roll up data. Whereas traditional OLAP systems simply apply aggregation functions on numerical measures in order to roll them up onto higher levels of abstraction, the merge operator works in a different way. Facts, which are only valid in a specific context, are associated with modules which represent different business scenarios. Therefore, also modules are only valid in a specific context. All modules which are valid in the applied context need to be merged. Then the merged module needs to be associated with the higher-level context. For example, if the levels *Level_Department_All*, *Level_Time_All* and *Level_Location_Continent* are parameters for the merge operator, all the modules which are valid in the context of *Location_Continent,* but also modules of more specific contexts need to be merged.

The second possibility to roll up data differs from the merge operator. While the merge operator seeks to combine all the information of the modules, the abstract operator focuses on a specific module. Due to the fact that those modules cover business scenarios, which maybe got merged with other modules by the merge operator, the modules may get very big and confusing. This is why the analyst should be able to group parts of the content using the abstract operator in order to get a better overview and to generate new insights into the business scenario.

4 OLAP Operators

In this chapter the implemented OLAP operators will be explained from the point of view of an analyst. Therefore, all operators are illustrated using a running example specifically developed for each operator in order to explain their functionalities in detail. Furthermore, the parameters and the usage of the operators will be explained and documented. The operators have been implemented using Java classes which provide different methods to configure the operators.

4.1 Slice/Dice

The main idea of this operator is to enable an analyst to select a specific point of interest for further analysis. Basically it is very similar to the slice and dice operators of traditional OLAP systems where certain parts of the base data are extracted. Therefore the base data is stored in a base repository and the extracted data is stored in a temp repository. The base data consists of different modules which need to be extracted. Furthermore it is important to extract all necessary information which is needed to perform analysis later on. This means it is not enough only to extract the content of the modules. It is also necessary to extract the relevant information of the dimensional model as well as the information generated by the CKR framework.

In the example shown in Figure 4 there exist three different modules asserted with the respective contexts. For the slice/dice operator the analyst needs to specify the dimension attributes of interest. If the analyst wants to select only those contexts related to *Department_All, Location_Europe* and *Time_All* the operator extracts contexts and their respective modules which are directly asserted to those dimension attributes or roll up to them. Also the contexts and the modules which exist on higher level of abstraction need to be extracted.

This means the operator extracts the contexts and their asserted modules *Department_Sales-Location_Austria-Time_Y2012-Q2* and *Department_Sales-Location_Germany-Time_Y2012-Q2* because *Location_Austria* and *Location_Germany* roll up to *Location_Europe*. The module *Department_Sales-Location_Japan-Time_Y2012Q-2* is not considered because *Location_Japan* does not roll up to *Location_Europe*.

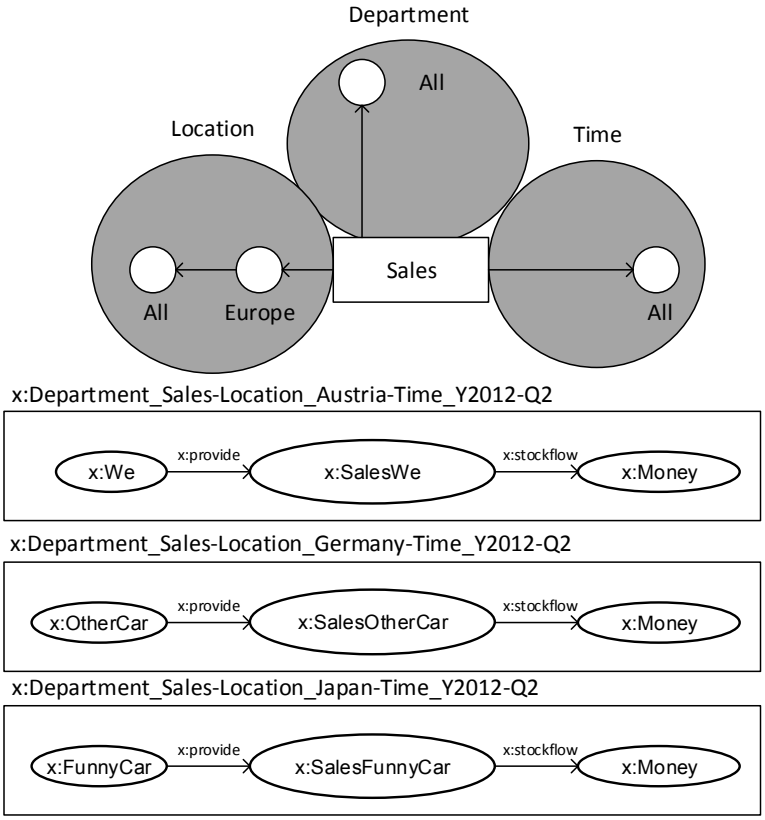

Figure 4: Example dataset for the slice/dice operator

As already mentioned the slice/dice operator also extracts the specific information of the dimensional model. If an analyst defines the dimension attribute value *Location_Europe*, all values need to be considered which roll up to *Location_Europe*, but also all values which *Location_Europe* rolls up to. Analogous to the extraction of modules and contexts explained before, this is necessary because an analyst may define *Location_Europe* but also wants to be able to do analysis based on *Cities*, *Countries*, *Continents* or *All*. This is why all information related to any of the defined dimension attributes also needs to be extracted from the dimensional model.

The CKR framework generates additional information about the dimensional model, contexts and modules (see Section 5.1 for further details). It needs to be ensured that these data also only includes information related to the defined dimension attributes.

Method	Parameter description	Datatype	Mandatory
setCoordinate	Property namespace	String	Yes
setCoordinate	Property name	String	Yes
setCoordinate	Dimension attribute namespace	String	Yes
setCoordinate	Dimension attribute name	String	Yes

Table 1: Slice/dice operator methods description

```
1   SliceDice sd = new SliceDice();
2
3   sd.setCoordinate(
4           configuration.getOlapModelNamespace(),
5           "hasTime",
6           configuration.getOlapModelNamespace(),
7           "Time_All");
8
9   sd. setCoordinate (
10          configuration.getOlapModelNamespace(),
11          "hasLocation",
12          configuration.getOlapModelNamespace(),
13          "Location_Europe");
14
15  sd. setCoordinate (
16          configuration.getOlapModelNamespace(),
17          "hasDepartment",
18          configuration.getOlapModelNamespace()
19          "Department_All");
20
21  sd.execute();
```

Listing 7: Usage of slice/dice operator with dimension attributes Time_All, Location_Europe and Department_All

Table 1 describes the method *setCoordinate* used to define the coordinates to be extracted from the OLAP cube. It uses four parameters in order to set the *property namespace, property name, dimension attribute namespace* and the *dimension attribute name.* For one coordinate the analyst is able to define one dimension attribute for a specific property. So if the analyst wants to extract data related to different dimension attributes of the same dimension, e.g. *Location_Europe* and *Location_Asia*, the operator needs to be executed twice. If no dimension attribute for a dimension is defined, it is assumed that the dimension attribute is the most granular one, e.g. *Location_All*. Listing 7 shows some sample code in order to configure the operator with

the dimension attributes *Time_All, Location_Europe* and *Department_All* as shown in Figure 4.

4.2 Merge

This operator is used to merge the content of different modules. Due to the fact that the slice/dice operator extracts all the relevant information into the temp repository, this operator only needs to access the temp repository. The advantage of this approach is that the base data does not get modified because it is isolated in a different repository. Two different variations have been implemented in order to do merging which will be described in the following sections.

4.2.1 Union

The first variation combines different modules independent of their content. This means that the content of the modules is not relevant for the result. The main idea of this variation is to merge the knowledge of different modules assigned to given lower-level contexts into a newly generated module which is then assigned to a higher-level context.

In the example shown in Figure 5a there exist different modules which are asserted with the respective contexts. The analyst needs to specify the levels to which the modules should be rolled up to. If the analyst wants to roll the modules up to the levels *Level_Department_Department, Level_Location_Continent* and *Level_Time_Year* all the modules which are asserted to a context which rolls up to those levels are merged into a new module. This means that the dimension attributes *Location_Austria*, *Location_Germany* and *Location_Japan* are rolled up to their assigned continent *Location_Europe* or *Location_Asia*. *Time_Y2012-Q2* is rolled up to *Time_Y2012*. As shown in Figure 5b the modules *Department_Sales-Location_Austria-Time_Y2012-Q2, Department_Sales-Location_Germany-Time_Y2012-Q2* will be merged to the module *Department_Sales-Location_Europe-Time_Y2012* and the module *Department_Sales-Location_Japan-Time_Y2012-Q2* will be merged to the module *Department_Sales-Location_Asia-Time_Y2012*. All merged modules need to be deleted. So the operator deletes the modules *Department_Sales-Location_Austria-Time_Y2012-Q2, Department_Sales-Location_Germany-Time_Y2012-Q2, Department_Sales-Location_Japan-Time_Y2012-Q2* and also the assertion between the deleted modules and their contexts.

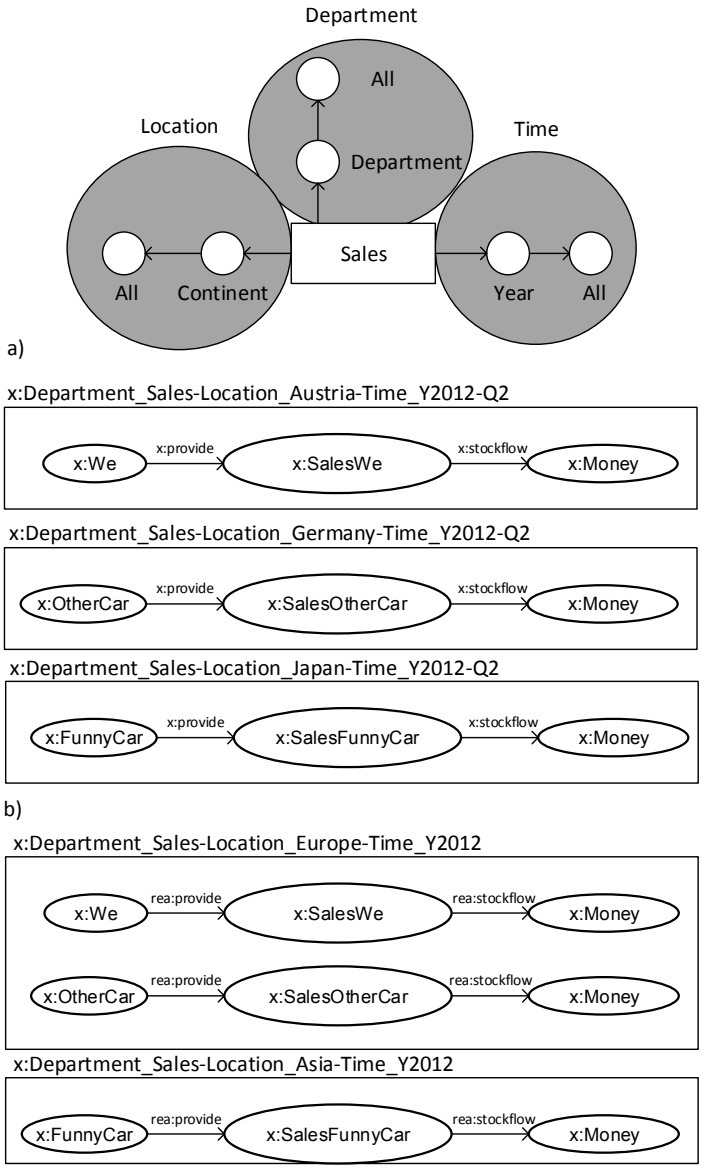

Figure 5: Example dataset (a) and result (b) of the merge operator (union)

An RDF graph is a set of triples. The result of merged named graphs does not contain possible duplicates. Depending on the type of analysis it may be important to know the number of duplicates. For the functionality of the abstract literal by source operator it is important to know the number of duplicate triples containing literals. For example, the triple *We revenue 10* may exist in different modules. As shown in Figure 6a the triple exists in two modules assigned to contexts with different dimension attributes *Location_Austria* and *Location_Germany*. This means in both countries there was a *revenue* of *10*. This knowledge need to be combined in a way the knowledge is preserved because otherwise the revenue on the level of *Level_Location_Continent* is not correct after the union operation.

A way to preserve this knowledge is to use the reification vocabulary of RDF. With this approach it is possible to save triples in a different representation in order to add additional information to it. This additional information may be the number of triples containing literals merged together as described in [10]. This enables to save the number of equal triples in the newly generated modules in order to preserve the information of the lower-level modules. As shown in Figure 6b the triple *We revenue 10* exists in two different modules which need to be merged. So in the newly generated module the information is added that this triple existed two times which is displayed by the number *2* in square brackets.

The analyst needs to configure the operator with the respective methods listed in Table 2. The method *setGranularity* is used to specify the *properties* of the dimension attributes and the *levels* which define the level of abstraction. It uses four parameters in order to set the *property namespace, property name, level namespace* and the *level name*. To define the namespace for the newly generated module the method *setGen eratedModuleNamespace* is used. As already mentioned there exist two different types of merging. The type needs to be set to *UNION* or *INTERSECTION* using the method *setMethod*. The concept of RDF reification may not make sense for every kind of data, so this functionality may be activated or deactivated by the use of the method *setReification*.

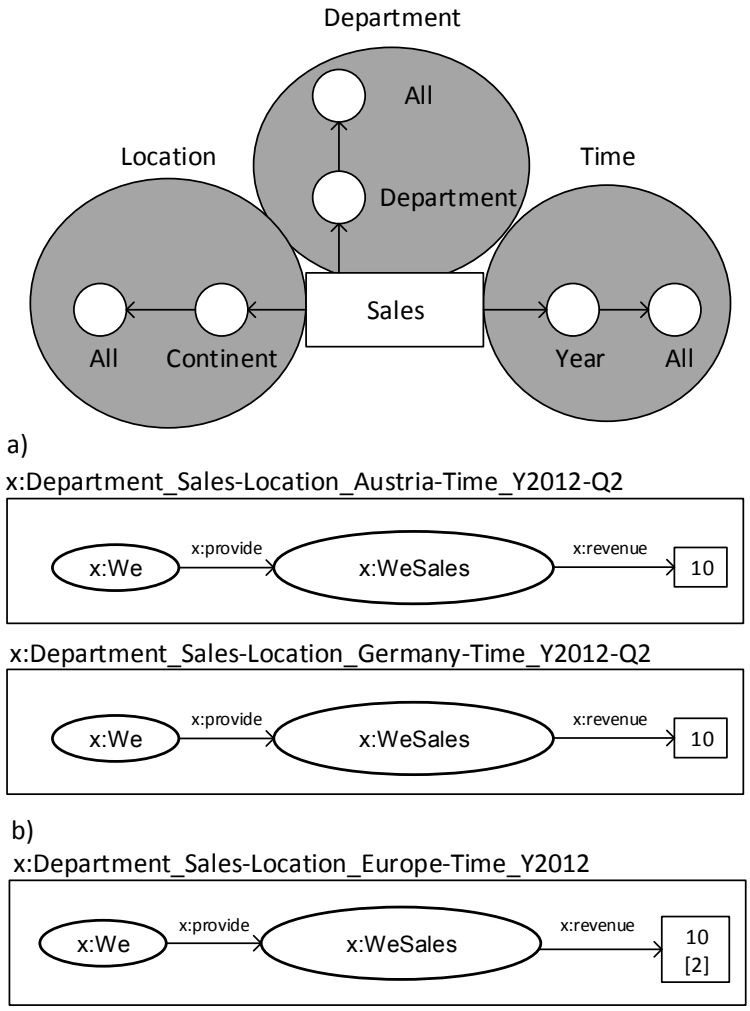

Figure 6: Example dataset (a) and result (b) of the merge operator (union) considering RDF reification

Listing 8 shows sample code in order to configure the operator as shown in Figure 6. The namespace for the newly generated resource is set in Lines 3-4. The levels *Level_Time_Year, Level_Location_Continent* and *Level_Department_Department* are set in Lines 6-22. The variant of the operator is set to *UNION* in Line 24. Line 25 configures the operator to support RDF reification.

Method	Parameter description	Datatype	Mandatory
setGranularity	Property namespace	String	Yes
setGranularity	Property name	String	Yes
setGranularity	Level namespace	String	Yes
setGranularity	Level name	String	Yes
setGeneratedModuleNamespace	Namespace of the generated module	String	Yes
setMethod	Method type	Enum	Yes
setReification	Reification	Boolean	Yes

Table 2: Merge operator methods description

```
1   Merge merge = new Merge();
2
3   merge.setGeneratedModuleNamespace(
4           configuration.getOlapModelNamespace());
5
6   merge.setGranularity(
7           configuration.getOlapModelNamespace(),
8           "hasTime",
9           configuration.getOlapModelNamespace(),
10          "Level_Time_Year");
11
12  merge.setGranularity(
13          configuration.getOlapModelNamespace(),
14          "hasLocation",
15          configuration.getOlapModelNamespace(),
16          "Level_Location_Continent");
17
18  merge.setGranularity(
19          configuration.getOlapModelNamespace(),
20          "hasDepartment",
21          configuration.getOlapModelNamespace(),
22          "Level_Department_Department");
23
24  merge.setMethod(MergeMethod.UNION);
25  merge.setDoReification(true);
26
27  merge.execute();
```

Listing 8: Usage of merge (union) operator with levels Level_Time_Year, Level_Location_Continent and Level_Department_Department

4.2.2 Intersection

This variant of the merge operator is similar to the functionality of the Union variant. In contrast to the Union variant the content of the modules is relevant. The intersection variant searches the content of the modules to find sets of triples which exist in every single module. If sets of equal triples are found, they are merged into newly generated modules.

As shown in Figure 7a there exist two different modules *Department_Sales-Location_Austria-Time_Y2012-Q2* and *Department_Sales-Location_Germany-Time_Y2012-Q2*. In both modules there exists a set of equal triples namely *We provide SalesWe* and *SalesWe stockflow Money*. This set of equal triples is merged into the newly generated module *Department_Sales-Location_Europe-Time_Y2012* shown in Figure 7b.

The RDF reification concept has also been implemented for the intersection variant. As already mentioned, the focus of this variant is to find sets of triples, which exist in all modules to be merged. For example, if an analyst wants to figure out which European companies do have a revenue of 10 it may not make sense for the analyst to know the number of modules this triple exists in. However, there may be scenarios where the analyst also wants to consider reification information for the intersection variant so it works analogous to the union variant. The parameters and the usage of the intersection variant are the same as it is shown in Table 2 and Listing 8. The only thing that needs to be changed is to set the method to *MergeMethod.INTERSECTION* instead of *MergeMethod.UNION*.

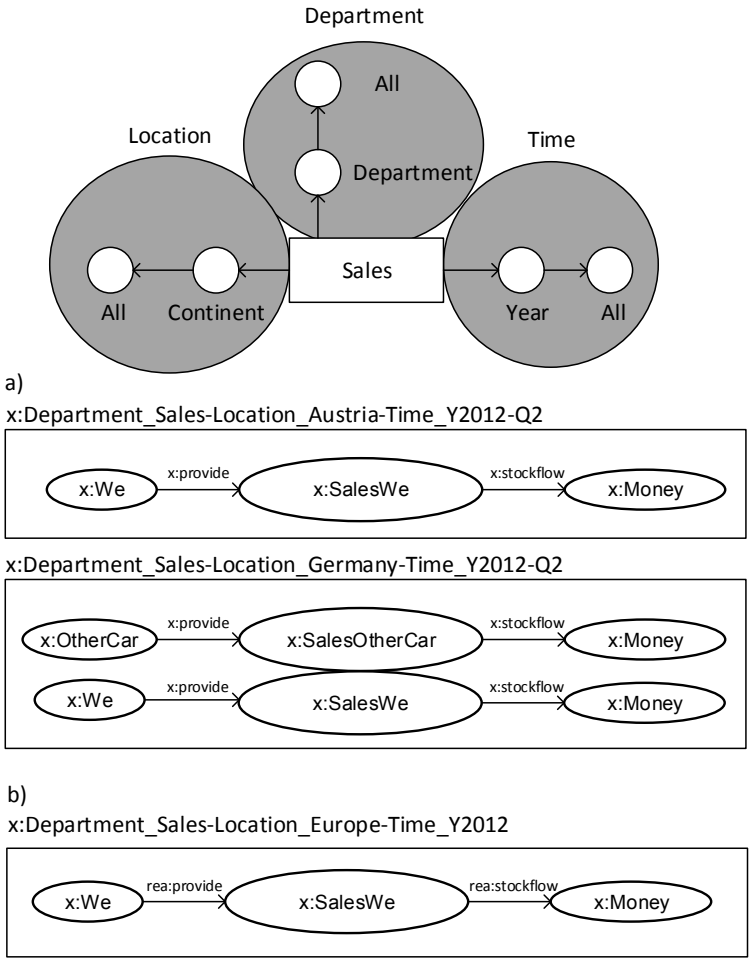

Figure 7: Example dataset (a) and result (b) of the merge operator (intersection)

4.3 Abstract

The focus of the different abstracts differs from the slice/dice and merge operator. Abstracts work on specific modules rather than dimension attributes or levels. Within those modules different grouping operations are executed. It replaces triples by more abstract triples. This may help an analyst to generate new insights into business.

Basically the abstracts can be divided into the types triple-generating, resource-generating and value-generating. Triple-generating abstracts generate new triples which were not present in the module before the execution of the abstract. Resource-generating abstracts generate new resources. This is done by the generation of a unique identifier to prevent the generation of duplicates. The last type of abstracts are value-generating abstracts which generate new literals. We only use literals representing numeric values. Note that a specific abstract may belong to more than one type. For example, an abstract of type value-generating also generates triples so it also of type triple-generating. Value-generating abstracts do not generate a unique identifier for the resources so they are not of type resource-generating.

In the following the different implemented abstracts are described. Note that the abstract literal by source operates only on literals, whereas the abstract by grouping abstract property by grouping and abstract property by source operate on resources represented by an IRI. Another important fact is that the abstracts may be configured by setting different properties which need to be present in order to execute the operation. Those properties do not need to be available directly within the corresponding module as they may are inherited from modules of higher-level contexts. For every Abstract operator an example dataset is presented. The resulting datasets show the result of the example dataset after execution of the respective operator.

4.3.1 Abstract By Grouping

With this operator specific resources can be replaced by other resources of the same module or by resources of inherited modules. This operator is of type triple-generating because the newly generated triples consist of resources already present directly in the module or in shared facts and so do not need to be generated.

Figure 8 shows the running example for this operator. Basically this example describes a business scenario of three different companies *We, FunnyCar* and *OtherCar*. Those companies belong to the same group *Company* which is specified by the *grouping* property. Note that for all abstracts the triples, containing a grouping property, remain unchanged. So *We grouping Company* will not be rolled up to *Company grouping Company*. For readability considerations, in the following the grouping properties are not part of the visualizations. *We, FunnyCar* and *OtherCar* all have a sister company and are a sub company of another company described by the properties *sisterCompanyOf* and *subCompanyOf*. Furthermore, those three companies

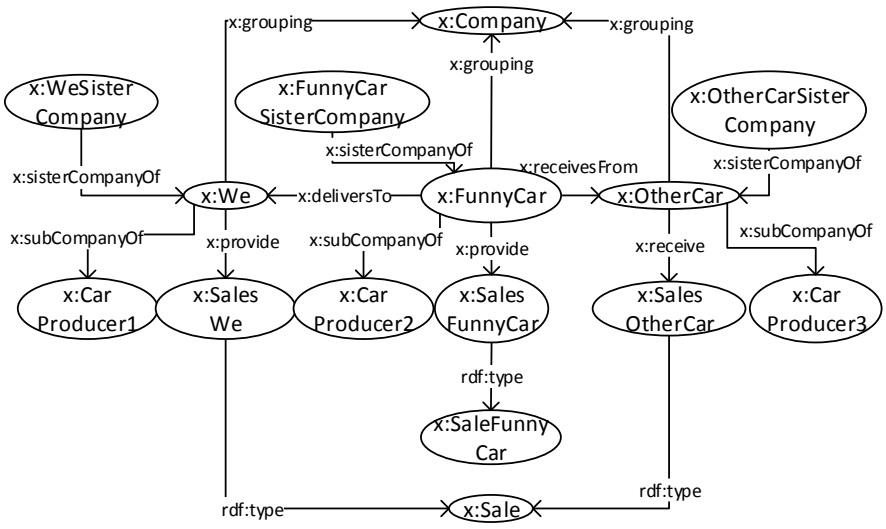

Figure 8: Example dataset for the abstract by grouping operator

provide or *receive Sales* of a specific *Type*. The properties *deliversTo* and *receives-From* describe properties which exist between *We, FunnyCar* and *OtherCar*.

The first example of this operator is to set the *grouping* property to *grouping*. All resources which have the same target resource with the same *grouping* property defined, should be replaced by the target resource. Figure 9 shows the result after execution of the operator on the dataset shown in Figure 8. The resources *We*, *FunnyCar* and *OtherCar* are replaced by *Company* because they have the same target resource *Company* defined by the same *grouping* property. The next step is to update all the properties which are incoming or outgoing to the resources which have been grouped. For example, the triple *WeSisterCompany sisterCompanyOf We* needs to be updated to *WeSisterCompany sisterCompanyOf Company*. Also the properties between the grouped resources need to be updated, e.g. *FunnyCar receivesFrom OtherCar* needs to be updated to *Company receivesFrom Company*.

The resources which need to be grouped may be specified in a more restrictive way by setting the *selection* property in addition to the *grouping* property. This defines that not all resources with the same grouping property should be grouped. In order to be considered the resources need to have the same *selection* property defined. Figure 10 shows the result of setting the *selection* property to *provide* in addition to the *grouping* property. This means that only the resources are replaced by the target of

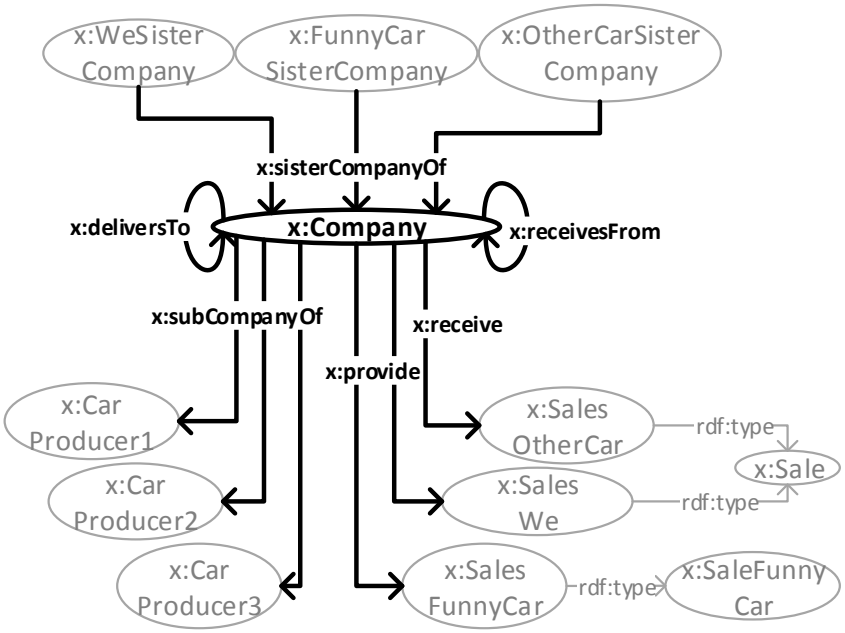

Figure 9: Resulting dataset after execution of the abstract by grouping operator with the grouping property set to grouping applied on Figure 8

the *grouping* property for which the *provide* property is defined. The operator replaces the resources *We* and *FunnyCar* with *Company* because for those resources the *provide* property is present. Also all incoming and outgoing properties need to be updated. For *OtherCar* no property *provide* exists so this resource is not replaced.

A different way to define the resources to group is to set the *selection resource type*. The focus is on the type of the resources connected to the target resources. With reference to the original dataset of Figure 8, the resource *SalesWe* needs to have a specific type in order to group the resource *We*. Figure 11 shows the result of setting the *selection resource type* to *Sale* in addition to the *grouping* property. The resources *SalesWe* and *SalesOtherCar* are of type *Sale* so only the corresponding resources *We* and *OtherCar* are grouped. This is not the case for *FunnyCar* because the resource *SalesFunnyCar* is of type *SaleFunnyCar*. Furthermore, all incoming and outgoing properties also need to be updated.

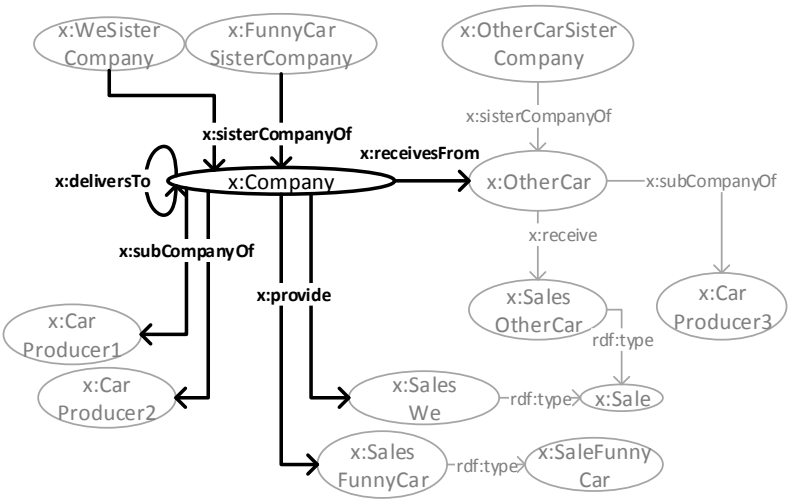

Figure 10: Resulting dataset after execution of the abstract by grouping operator with the grouping property set to grouping, the selection property set to provide applied on Figure 8

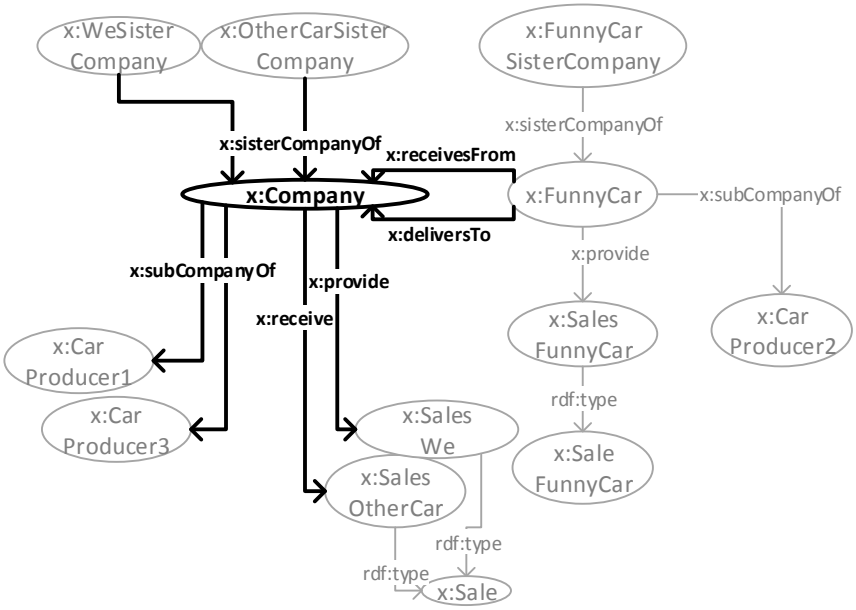

Figure 11: Resulting dataset after execution of the abstract by grouping operator with the grouping property set to grouping, the selection resource type set to Sale applied on Figure 8

The abstracts also need to consider sets of equal triples which represent numeric values. To illustrate this specific case the example from Figure 8 gets extended with literals which represent numeric values. Figure 12 shows the example dataset including reification information. The resource *FunnyCar* has a *revenue* of *10* whereas for the resources *We* and *Company* reification information already exists. Those information may was defined manually or generated by the merge operator. This means that the Abstract operator has to be able to handle both types of information in order calculate the correct number of sets of equal triples containing literals. Figure 13 shows the resulting dataset after execution of the operator on the example dataset. If, for example the *grouping* property is set to *grouping* the resources *We, FunnyCar* and *OtherCar* needs to be replaced by *Company*. So the right number of literals needs to be calculated. The triples *We revenue 10* and *Company revenue 10* both exist two times whereas *FunnyCar revenue 10* exists only once. After the replacement of the resources by their grouping resource *Company,* the triple *Company revenue 10* exists five times. Note this behavior is analogous for the other abstracts abstract property by grouping and abstract property by source so it will only be explained once.

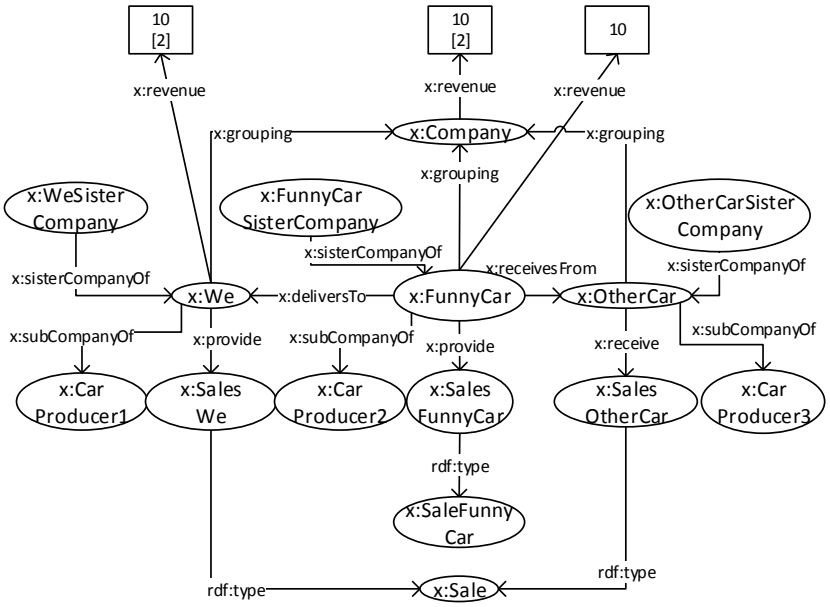

Figure 12: Example dataset including reification information for the abstract by grouping operator

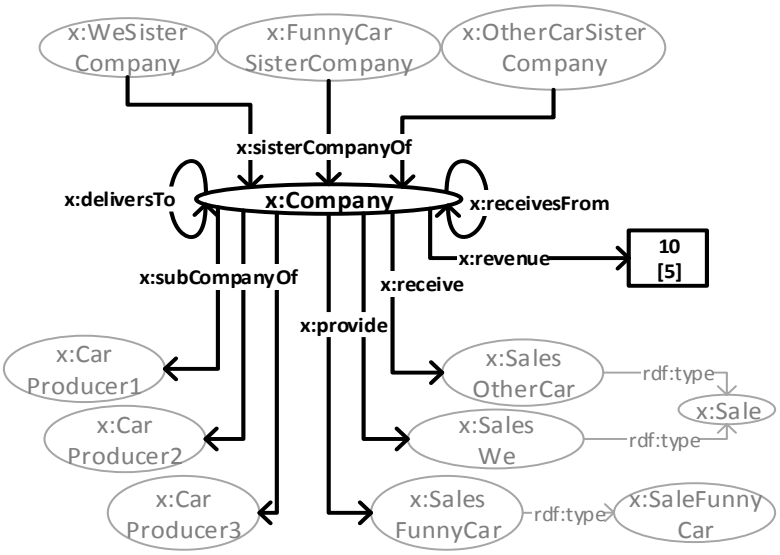

Figure 13: Resulting dataset after execution of the abstract by grouping operator with the grouping property set and reification enabled applied on Figure 12

Table 3 shows the description of the methods provided by the abstract by grouping operator. In order to execute the operator, it is mandatory to set the *grouping property* and the *graph*. Due to the fact that modules are defined as named graphs, this is equal to the definition of the model the operator should be executed on. All other properties do not need to be set. Note that for all abstracts it is possible to use the options in combination. Listing 9 shows the configuration of the operator considering reification (Line 4) with the *grouping property* (Lines 10-12), *selection property* (Lines 14-16) and *selection resource type* (Lines 18-20) set.

Method	Parameter description	Datatype	Mandatory
setGroupingProperty	Property namespace	String	Yes
setGroupingProperty	Property name	String	Yes
setSelectionProperty	Property namespace	String	No
setSelectionProperty	Property name	String	No
setSelectionResourceType	Type namespace	String	No
setSelectionResourceType	Type name	String	No
setGraph	Namespace of the module	String	Yes
setGraph	Name of the module	String	Yes
setReification	Reification	Boolean	No

Table 3: Abstract by grouping operator methods description

```
1  AbstractByGrouping abstrByGrouping =
2                          new AbstractByGrouping();
3
4  abstrByGrouping.setReification(true);
5
6  abstrByGrouping.setGraph(
7                  configuration.getOlapModelNamespace(),
8                  graph);
9
10 abstrByGrouping.setGroupingProperty(
11          "http://www.semanticweb.org/schnepf/ontology#",
12          "grouping");
13
14 abstrByGrouping.setSelectionProperty(
15          "http://www.semanticweb.org/schnepf/ontology#",
16          "provide");
17
18 abstrByGrouping.setSelectionResourceType(
19          "http://www.semanticweb.org/schnepf/ontology#",
20          "Sale");
21
22 abstrByGrouping.execute();
```

Listing 9: Usage of abstract by grouping operator considering reification with the grouping property, selection property and selection resource type set

4.3.2 Abstract Property By Grouping

This operator is similar to the abstract by grouping operator but provides more comprehensive configuration possibilities. For example, it is possible to explicitly define properties which should be set to the target resource of the grouping property.

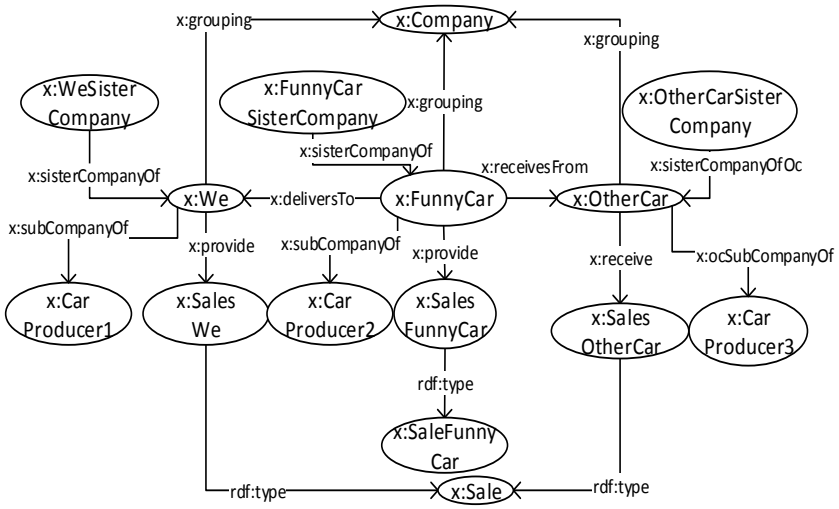

Figure 14: Example dataset for the abstract property by grouping operator

Figure 14 shows the running example for this operator. Basically this example is similar to the one shown in Figure 8. The differences are the incoming and outgoing properties of the resource *OtherCar*. Those specific properties are necessary to show all functionalities of this operator.

To define the properties which should be set to the target resource of the grouping property different variations are possible. The first one is to set the direction of the properties to be grouped. So it is possible to define that only those properties should be set to the target resource of the grouping property which are incoming or outgoing using the property *grouped property direction*. Figure 15 shows the resulting dataset after execution of the operator on the example dataset with the grouping property *grouping* and the grouped property direction set to *incoming*. The resources *We, FunnyCar* and *OtherCar* are replaced by *Company*. In contrast to the abstract by grouping operator only those properties are set to *Company* which are incoming properties for the resources *We*, *FunnyCar* and *OtherCar*. All other properties remain unchanged. This is also possible for the opposite direction in order to group only those properties which are outgoing of the resources *We*, *FunnyCar* and *OtherCar*. If no specific direction is set the operator considers both incoming and outgoing properties.

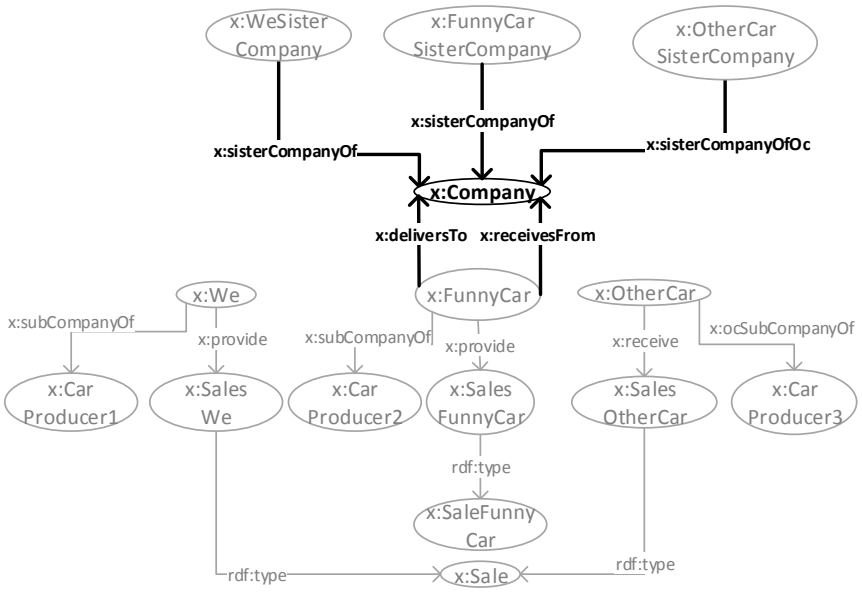

Figure 15: Resulting dataset after execution of the abstract property by grouping operator with the grouping property set to grouping, the grouped property direction set to incoming applied on Figure 14

Another possibility to specify the properties to be grouped is to set a specific *grouped property*. Figure 16 illustrates the result after execution of the Abstract operator on the example dataset. Additionally to the *grouping* property the *grouped property* is set to *sisterCompanyOf*. So only the target of this specific property is set to *Company*, all other properties remain unchanged.

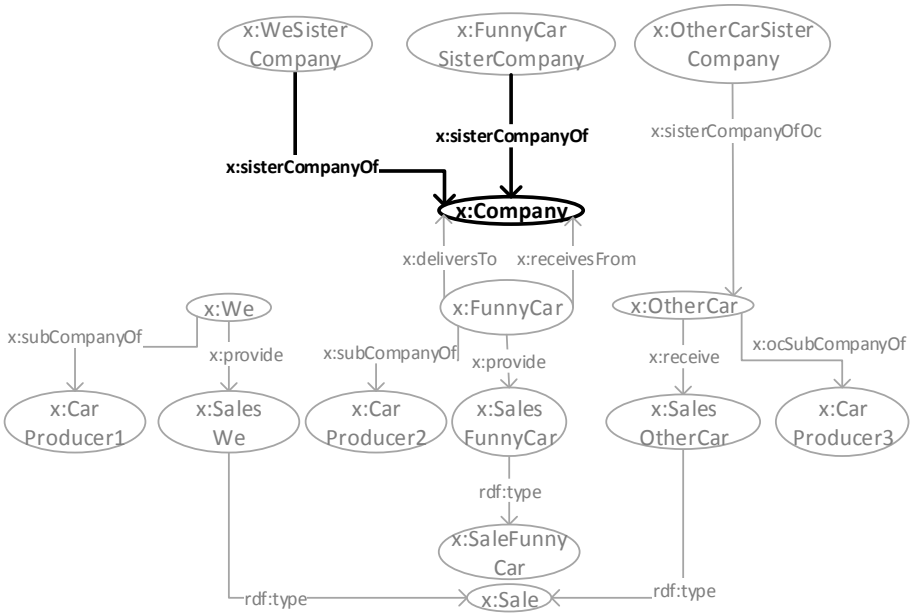

Figure 16: Resulting dataset after execution of the abstract property by grouping operator with the grouping property set to grouping, the grouped property set to sisterCompanyOf applied on Figure 14

Table 4 shows the available methods and the parameters of the abstract property by grouping operator. They are equal to the ones of the abstract by grouping operator except for the method *setGroupedProperty* and *setGroupedPropertyDirection*. Note that these parameters are not mandatory. Listing 10 illustrates the usage of the operator by setting the *grouping* (Lines 8-10) and the *grouped* (Lines 12-14) property in order to generate the result shown in Figure 16.

Method	Parameter description	Datatype	Mandatory
setGroupingProperty	Property namespace	String	Yes
setGroupingProperty	Property name	String	Yes
setSelectionProperty	Property namespace	String	No
setSelectionProperty	Property name	String	No
setSelectionResourceType	Type namespace	String	No
setSelectionResourceType	Type name	String	No
setGroupedProperty	Property name	String	No
setGroupedProperty	Property namespace	String	No
setGroupedPropertyDirection	Direction of the resources to be grouped	Enum	No
setGraph	Namespace of the module	String	Yes
setGraph	Name of the module	String	Yes
setReification	Reification	Boolean	No

Table 4: Abstract property by grouping methods description

```
1  AbstractPropertyByGrouping abstrPropByGrouping =
2                          new AbstractPropertyByGrouping();
3
4  abstrPropByGrouping.setGraph(
5           configuration.getOlapModelNamespace(),
6           graph);
7
8  abstrPropByGrouping.setGroupingProperty(
9           "http://www.semanticweb.org/schnepf/ontology#",
10          "grouping");
11
12 abstrPropByGrouping.setGroupedProperty(
13          "http://www.semanticweb.org/schnepf/ontology#",
14          "sisterCompanyOf");
15
16 abstrPropByGrouping.execute();
```

Listing 10: Usage of abstract property by grouping operator with the grouping property and the grouped property set

4.3.3 Abstract Property By Source

While the abstract by grouping and the abstract property by grouping operator replace the source of a triple, the abstract property by source replaces the target of a triple. For example, the target of triples with a source of a specific grouping and specific properties may be grouped. To group target resources new resources need to be generated using unique IDs. This abstract is of both types *triple-generating* and *resource-generating*.

Figure 17 shows the running example for this operator. Basically there exist three different companies *We, FunnyCar* and *OtherCar* with the same *grouping property* as in the examples before. The companies *provide* or *receive* specific *Sales* of different types *Sale* or *SaleFunnyCar*. Furthermore a relation between the *Sales* is defined by the properties *relatedToFc* and *relatedToOc*. The sales may generate stock flows which are represented by the properties *stockflow* and *stockFlowOc*.

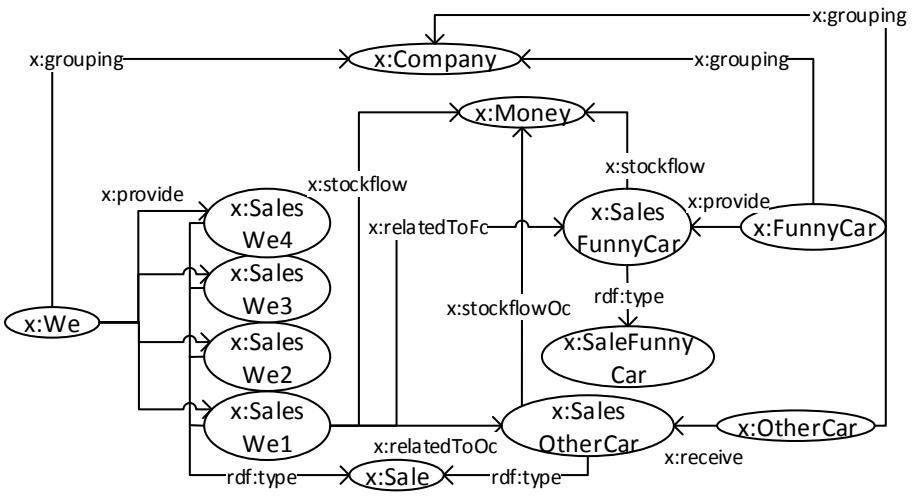

Figure 17: Example dataset for the abstract property by source operator

Figure 18 shows the resulting dataset after executing the operator on the example dataset with the *grouping property* set to *grouping*. There are three companies *We, FunnyCar* and *OtherCar* which have the same *grouping.* Every single company *provides* or *receives Sales.* So three new resources need to be generated in order to group all target resources where the sources have the same grouping. The knowledge which

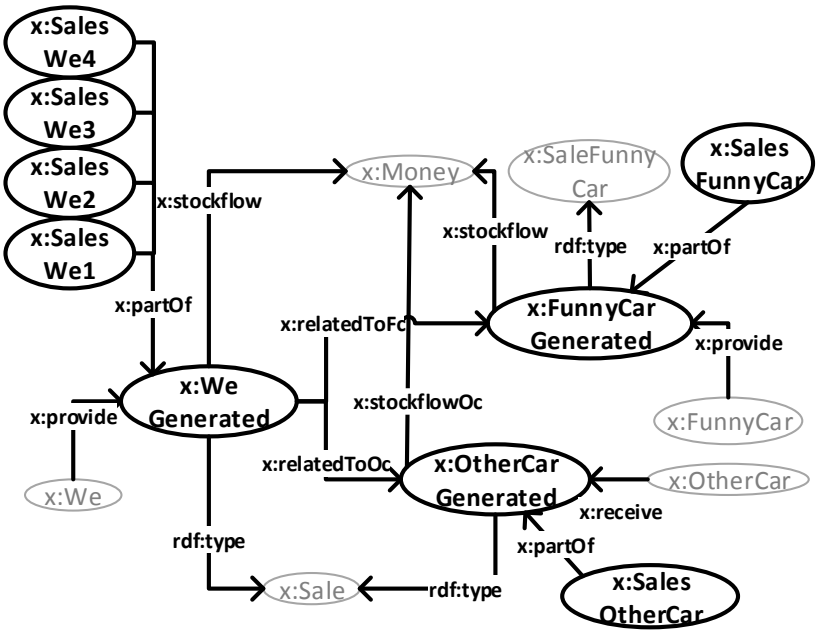

Figure 18: Resulting dataset after execution of the abstract property by source operator with the grouping property set to grouping applied on Figure 17

sales are grouped to the newly generated resource should persist. Therefore the analyst is able to define a property which maps existing sales to the newly generated sale. Analogous to the already explained abstract variants all incoming and outgoing properties need to be set to the newly generated resource.

As shown in Table 5 this operator provides the same methods as the abstract variants already explained. In addition this operator defines two new methods. The first one is *setPartitionProperty* in order to map the grouped resources to the newly generated aggregated resource. In the example shown by Figure 18 the *partition* property was set to *partOf*. The second method is *setGeneratedResourceNamespace* which enables the analyst to define a specific namespace for the newly generated resource. The name of the resource is concatenated by a unique ID and the name of the source. If the *selectionProperty* or the *groupedProperty* is set, the names of those properties also are concatenated to the name of the newly generated resource. To keep the examples readable the generated unique ID is not part of the visualization. Analogous to

Method	Parameter description	Datatype	Mandatory
setGroupingProperty	Property namespace	String	Yes
setGroupingProperty	Property name	String	Yes
setSelectionProperty	Property namespace	String	No
setSelectionProperty	Property name	String	No
setSelectionResourceType	Type namespace	String	No
setSelectionResourceType	Type name	String	No
setGroupedProperty	Property name	String	No
setGroupedProperty	Property namespace	String	No
setGroupedPropertyDirection	Direction of the resources to be grouped	Enum	No
setPartitionProperty	Property name	String	Yes
setPartitionProperty	Property namespace	String	Yes
setGeneratedResourceNamespace	Namespace of the resource to be generated	String	Yes
setGraph	Namespace of the module	String	Yes
setGraph	Name of the module	String	Yes
setReification	Reification	Boolean	No

Table 5: Abstract property by source methods description

the other abstraction variants it is possible to use all properties in combination. Listing 11 shows the usage of the operator. The namespace of the newly generated resource is set in Lines 8-9. The grouping property, selection property, partition property, grouped property and the selection resource type are set in Lines 11-29.

```
1   AbstractPropertyBySource abstrPropertyBySource=
2      new AbstractPropertyBySource();
3
4   abstrPropertyBySource.setGraph(
5               configuration.getOlapModelNamespace(),
6               graph);
7
8   abstrPropertyBySource.setGeneratedResourceNamespace(
9            "http://www.semanticweb.org/schnepf/ontology#");
10
11  abstrPropertyBySource.setGroupingProperty(
12             "http://www.semanticweb.org/schnepf/ontology#",
13             "grouping");
14
15  abstrPropertyBySource.setSelectionProperty(
16             "http://www.semanticweb.org/schnepf/ontology#",
17             "provide");
18
19  abstrPropertyBySource.setPartitionProperty(
20             "http://www.semanticweb.org/schnepf/ontology#",
21             "partOf");
22
23  abstrPropertyBySource.setGroupedProperty(
24             "http://www.semanticweb.org/schnepf/ontology#",
25             "stockFlow");
26
27  abstrPropertyBySource.setSelectionResourceType(
28             "http://www.semanticweb.org/schnepf/ontology#",
29             "Sale");
30
31  abstrPropertyBySource.execute();
```

Listing 11: Usage of abstract property by source operator with the grouping property, selection property, partition property, grouped property and selection resource type set

4.3.4 Abstract Literal By Source

This abstract is the only one which operates on literals which represent numeric values. So this operator generates new literals by executing different aggregate functions on existing literals. It is of type triple-generating because it generates triples which did not exist before. It also is of type value-generating because of the generation of new aggregated literals. Note that this operator is able to handle only literals which represent numeric values.

Figure 19 shows the running example for this operator. *We, FunnyCar* and *OtherCar* provide different sales with literals asserted by the properties *revenue* and *sales*. With this operator it is possible to define the properties of the literals the operator should consider and the aggregation function which should be executed on the literals.

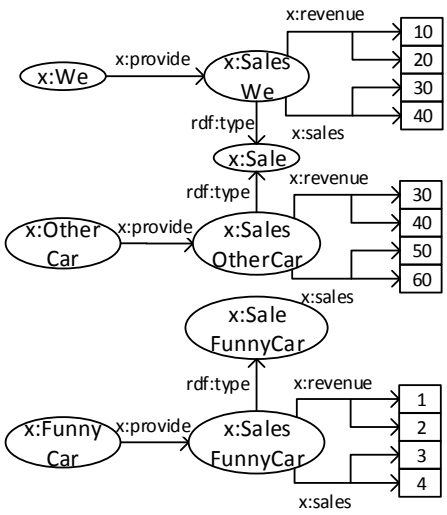

Figure 19: Example dataset for the abstract literal by source operator

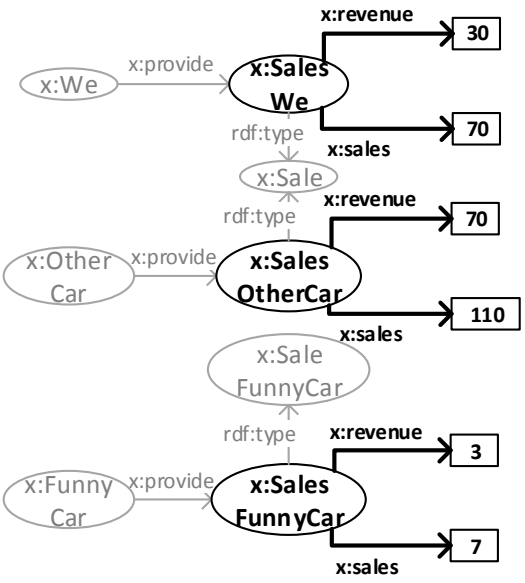

Figure 20: Resulting dataset after execution of the abstract literal by source operator with the aggregate function set

The first example illustrated in Figure 20 shows the result after setting the *aggregate function* to *SUM*. The operator sums up the numeric values of the literals by their respective source and property. For example, in reference to the example dataset shown in Figure 19 the values *10* and *20* are summed up to *30* because of the same source *SalesWe* and the same property *revenue*.

As already mentioned it is also possible to define the literals which should be summed up in a more specific way. Figure 21 shows the result after executing the operator on the example dataset with the *aggregate property* set to *revenue* and the *aggregate function* set to *SUM*. This has the effect that only the literals with the same source and a specific property are summed up. The literals of *SalesWe* with the property *revenue* are summed up to *30* whereas the literals with the property *sales* remain unchanged.

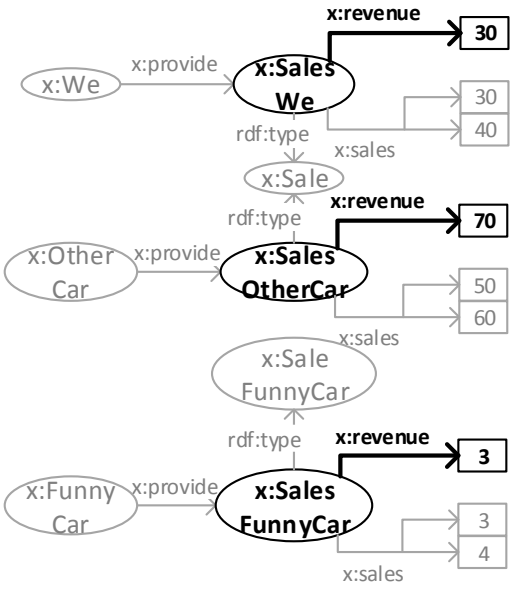

Figure 21: Resulting dataset after execution of the abstract literal by source operator with the aggregate function set to SUM, the aggregate property set to revenue applied on Figure 19

Another way is to set the resource type of the source. Figure 22 shows the result after executing the operator on the example dataset with the *selection resource type* set to *Sale* and the *aggregate function* set to *SUM*. This means that only those literals are summed up with a source of type *Sale*. The literals of *SalesFunnyCar* remain unchanged because the type of *SalesFunnyCar* is *SaleFunnyCar*.

The merge operator and the other abstract variants support the concept of RDF reification, so they generate knowledge on how often a literal exists instead of simply overwriting the existing triples. Because of that the abstract literal by source operator needs to support existing literals as well as literals stored by the use of RDF reifications. Figure 23 shows the example dataset including reification information. The triple *SalesWe revenue 10* exists multiple times which is defined by the number *2* in square brackets. If the literals of the resource *SalesWe* and the property *revenue* needs to be summed up, the newly generated literal is *40* instead of *30*.

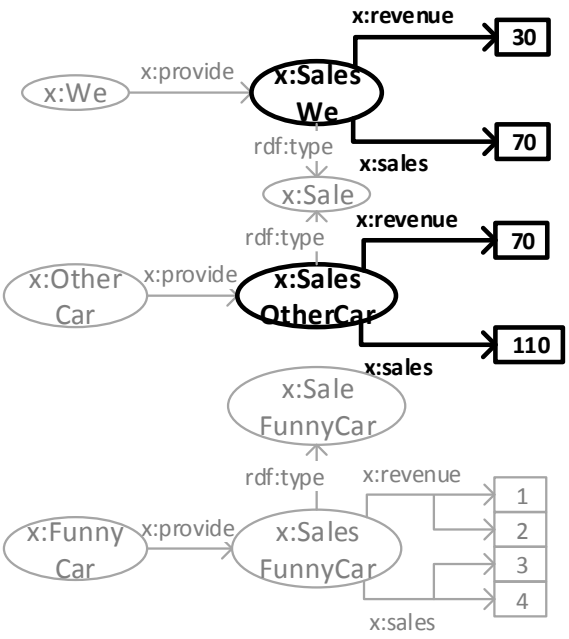

Figure 22: Resulting dataset after execution of the abstract literal by source operator with the aggregate function set to SUM, the selection resource type set to Sale applied on Figure 19

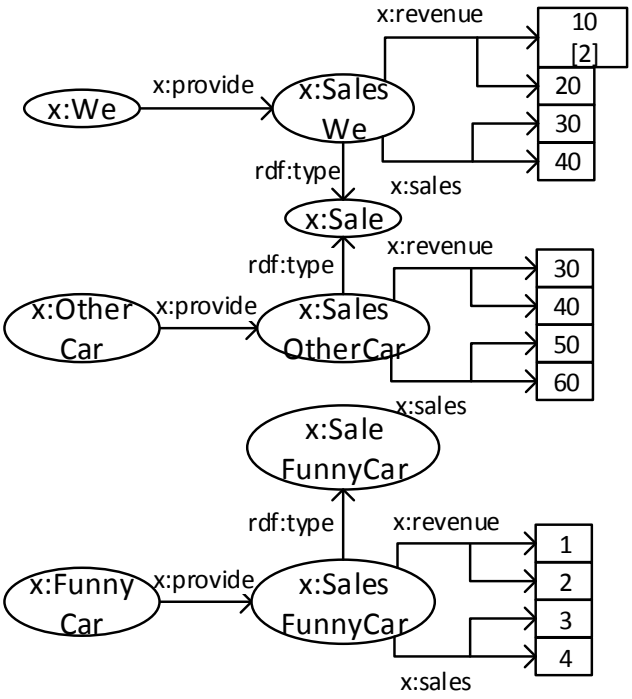

Figure 23: Example dataset including reification information for the abstract literal by source operator

Table 6 shows the different properties of the operator. The only properties which are mandatory are *graph* and *aggregate function*. Equal to the other types of abstracts it is possible to combine the different properties. To generate the result of Figure 20 it is necessary to configure the operator as shown in Listing 12. In Line 8 the *aggregate function* is set to *SUM*. Lines 10-12 set the *aggregate property* to *revenue*.

Method	Parameter description	Datatype	Mandatory
setAggregateFunction	Aggregate function which should be executed	String	Yes
setAggregateProperty	Property name	String	No
setAggregateProperty	Property namespace	String	No
setSelectionResourceType	Type namespace	String	No
setSelectionResourceType	Type name	String	No
setGraph	Namespace of the module	String	Yes
setGraph	Name of the module	String	Yes
setReification	Reification	Boolean	No

Table 6: Abstract literal by source methods description

```
1   AbstractLiteralBySource abstrLiteralBySource =
2                   new AbstractLiteralBySource();
3
4   abstrLiteralBySource.setGraph(
5                       configuration.getOlapModelNamespace(),
6                       graph);
7
8   abstrLiteralBySource.setAggregateFunction("SUM");
9
10  abstrLiteralBySource.setAggregateProperty(
11      "http://www.semanticweb.org/schnepf/ontology#",
12      "revenue");
13
14  abstrLiteralBySource.execute();
```

Listing 12: Usage of abstract literal by source operator with the aggregate function and the aggregate property set

5 Implementation

In this chapter different details of the API are explained. First, the system architecture is shown by explaining the parts interacting with each other. Secondly, the software architecture of the API is shown by a class diagram in order to present an overview about the different classes and their structure. Furthermore, the different SPARQL queries of the operators are explained. Note that the queries are generated dynamically depending on the parameters set.

5.1 System Architecture

Figure 24 shows the system architecture of the API. The base data includes the dimensional model, the assertion of the contexts to the modules as well as the definition of the modules. The base data is loaded into the base repository, an instance of the CKR framework implemented in Sesame. This is where all the information of the base data is analyzed and new information is generated and saved in a materialized way. Note that it is not necessary to use the CKR framework here. The API, however, expects the structure of the base data to be exactly as the base data generated by the CKR framework. Note that we use the CKR framework in combination with a custom ruleset provided by Loris Bozzato of the Fondazione Bruno Kessler specifically for our purposes, based on the materialization calculus for CKR [16].

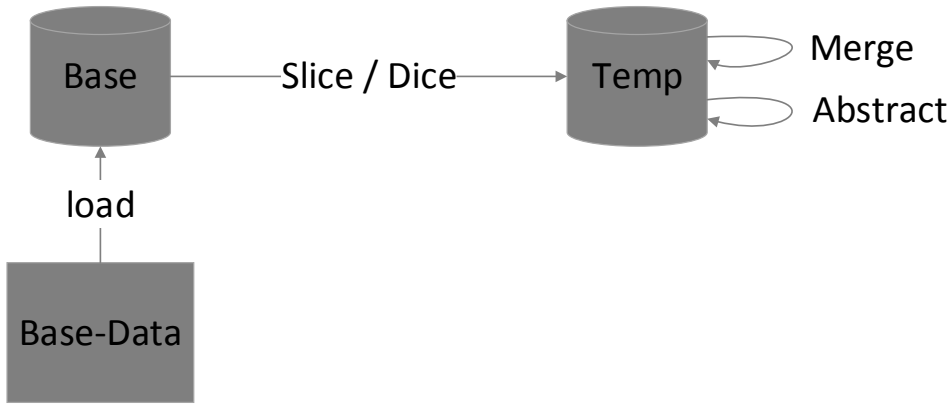

Figure 24: System architecture of the API

Based on the defined dimension attributes of the dimensional model, the CKR framework generates different contexts for every combination of dimension attributes. This means that for the dimension attributes *Department_All, Location_All, Location_Europe, Time_All* and *Time_Y2013* the contexts *Ctx-Department_All-Location_All-Time_Y2013, Ctx-Department_All-Location_Europe-Time_Y2013, Ctx-Department_All-Location_Europe-Time_All, Ctx-Department_All-Location_All-Time_All* are generated.

Another important functionality of CKR framework is knowledge propagation. It is important to know which modules of contexts on higher levels have modules which are also valid for contexts on lower level of abstraction. This is done by saving information about the relation of the modules similar to the relation of the different levels and dimension attributes. The CKR framework uses the *derivedFrom* property in order to define which module derives knowledge from another module of a context on a higher level of abstraction. The *derivedFrom* property does not assign contexts to modules directly. The CKR framework uses the *closureOf* property to assign a generated graph to a context. This graph saves all information needed for knowledge propagation. For example, for the *Ctx-Department_All-Location_Europe-Time_Y2013* the statements from Listing 13 are generated. The *generatedGraph* stands for a graph IRI which is generated by the CKR framework. As shown in the example it is so possible to select all the modules which are relevant for a specific context, those who are directly asserted and those who are assigned to contexts on higher level of abstraction.

```
1   generatedGraph closureOf       :Ctx-Department_All-
2                                         Location_Europe-Time_Y2013.
3   generatedGraph derivedFrom     :Module-Department_All-
4                                         Location_Europe-
5                                         Time_Y2013.
6   generatedGraph derivedFrom     :Module-Department_All-
7                                         Location_All-Time_All.
```

Listing 13: Knowledge propagation information

The slice/dice operator copies the data from the base repository into the temp repository. The merge and abstract operators are executed on the temp repository so the data in the base repository remains unchanged.

5.2 Software Architecture

Generally the API is a SPARQL-based implementation because all operators have been implemented using SPARQL update statements only. We use the Apache Jena[6] framework to handle those update statements. One important fact while implementing the API was to design the classes in a way they are independent of the used framework. This has the advantage that the API can be adapted for different frameworks only by changing few classes. Due to the fact that the operators are SPARQL updates executed against repositories, in principle all frameworks may be used which support the execution of SPARQL updates on remote repositories. The suitability of a framework depends on its interpretation of the default graph. It needs to interpret the default graph as the union of all named graphs. So it is possible to query one single graph in order to receive data from all named graphs.

Figure 25 shows the structure of the classes in an UML class diagram. In the upper left corner there is the abstract class *Statement* containing the method *execute*. All operators are specific statements and defined as subclasses of *Statement*. In those subclasses the SPARQL updates are generated and executed.

The *Statement* class uses the class *RepositoryConnector* in order to connect to specific repositories where the statements are executed. To enable a *Statement* to connect to a repository a *RepositoryConnector* needs to be set by using the method *setRepositoryConnector*. For every specific type of repository a specific subclass of *RepositoryConnector* needs to be created. The API is able to connect to repositories of type Sesame[7] and Jena with the respective classes *SesameRepositoryConnector* and *JenaRepositoryConnector*. Those classes define methods in order to execute statements directly on the repositories. So if a different repository type needs to be used it is only necessary to design a new subclass of *RepositoryConnector*. This class needs to define methods to execute the statements using a framework which is able to connect to the specific type of repository. The *RepositoryConnector* uses the *Configuration* class in order to store specific properties. The API offers the possibility to read the properties from a Java Properties File which stores the properties as key/value pairs. The class *PropertiesFileConfigurationFactory* directly reads the properties from a properties file. The API is able to read the properties from arbitrary sources. If an additional possibility should be implemented to read the properties from other sources it is only necessary to design a new subclass of *ConfigurationFactory*.

---

[6] https://jena.apache.org/
[7] http://rdf4j.org/

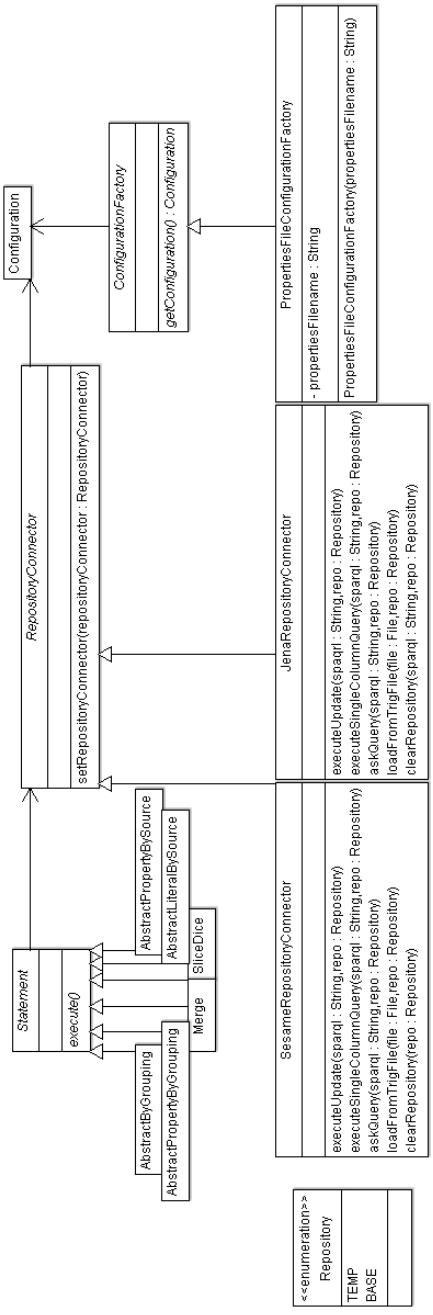

Figure 25: Class diagram

5.3 SPARQL Updates

This section describes how the operators have been implemented using SPARQL updates. There are a lot of RDF Frameworks which offer different functionalities for implementation. For example, with Jena it is possible to load RDF graphs into a Jena Model which then can be parsed and manipulated using Jena API directly. As most classes in the OLAP API should be independent from the framework used, the focus was on implementing the operators only by using SPARQL updates. Note that the slice/dice and merge operators are able to handle arbitrary amounts of dimensions.

5.3.1 Slice/Dice

As already mentioned this operator is used to copy the interesting data from the base repository into the temp repository. Therefore it uses different updates which will be explained in the following. Those updates are executed against the temp repository and are dynamically generated based on the configuration illustrated in Listing 7.

5.3.1.1 Contexts with asserted modules

The update shown in Listing 14 selects all contexts and the content of the asserted modules in order to insert those triples into the *global* graph of the temp repository (Lines 1-6). With the *SERVICE* statement in Line 8 it is possible to execute the query against an arbitrary SPARQL endpoint. In this query the *base* repository is defined as SPARQL endpoint because the data needs to be fetched from there. The assertion of *contexts* and *modules* is done by the property *hasAssertedModule* (Line 10). The modules are defined as graphs, so in Line 11 all triples *s p o* of those graphs *m* need to be selected which represent the content of the modules. Every context is defined by dimension attributes *d0 d1 d2* asserted by the respective properties *hasDepartment*, *hasLocation* and *hasTime* (Lines 12-14*)*. Lines 15-42 select the dimension attributes which are related to dimension attributes defined by the analyst. For example, Lines 24-32 query all dimension attributes *d1* which roll up to *Location_Europe* as well as the *d1* which are on higher level of abstraction *Location_Europe* rolls up to. This is because an analyst may want to copy all information which is valid in *Location_Europe* but also wants to be able to do further analytics on a higher level of abstraction like on *Level_Location_All*. Lines 16-18 and 20-22 of Listing 14 define independent subqueries of their own scope joined by the *UNION* keyword. The braces used in Line 15 and 23 also define subqueries. So subqueries may be connected by the

```
1  INSERT {
2   GRAPH ckr:global{
3    ?c :hasAssertedModule ?m.
4   }
5   GRAPH ?m { ?s ?p ?o }
6  }
7  WHERE {
8   SERVICE <http://localhost:50000/repositories/Base>{
9    SELECT distinct ?c ?m ?s ?p ?o WHERE {
10    ?c :hasAssertedModule ?m.
11    GRAPH ?m {?s ?p ?o}.
12    ?c :hasDepartment ?d0.
13    ?c :hasLocation ?d1.
14    ?c :hasTime ?d2.
15    {
16     {
17      ?d0 :directlyRollsUpTo* :Department_All.
18     }
19     UNION
20     {
21      :Department_All :directlyRollsUpTo* ?d0.
22     }
23    }
24    {
25     {
26      ?d1 :directlyRollsUpTo* :Location_Europe.
27     }
28     UNION
29     {
30      :Location_Europe :directlyRollsUpTo* ?d1.
31     }
32    }
33    {
34     {
35      ?d2 :directlyRollsUpTo* :Time_All.
36     }
37     UNION
38     {
39      :Time_All :directlyRollsUpTo* ?d2.
40     }
41    }
42   }
43  }
44 }
```

Listing 14: Update statement for the insertion of contexts and asserted modules with dimension attributes Department_All, Location_Europe and Time_All

usage of *UNION* or without any keyword. The difference is that with the *UNION* keyword the query engine does not match the values of the resources for equality and only joins the sets. This is comparable with an outer join in SQL. If the *UNION* keyword is not used, the engine checks the values for equality and merges the results

where equal resources exist. By the use of SPARQL property paths[8], as it is shown by the property *directlyRollsUpTo** (Line 17), it is possible to declare that the subject does not need to have the object directly assigned by the property, also a transitive closure is possible. So there may be an arbitrary amount of resources between the subject and the object assigned by the property *directlyRollsUpTo.*

5.3.1.2 Meta-information of the dimensional model

Listing 15 illustrates the update statement to insert parts of the dimensional model into the temp repository. Due to the fact that the analyst is able to define a point of interest with the slice/dice operator, it is necessary that only the interesting information of the dimensional model is extracted. This means that there should not be a dimension attribute *Location_Asia* if the analyst specifies *Location_Europe.* Note that some parts of the information to be copied depends on dimension attributes specified by the analyst. Due to the fact that the number of triples depending on dimension attributes is low, the query selects all dependent triples in Lines 14-32. With the MINUS statement (Line 13) it is now possible to define that none of the triples selected in Lines 14-32 should be part of the result. So only those triples are part of the result which are independent from the dimension attributes. For example, all triples *s p o* of the *global* graph (Lines 9-12) which are not asserted to a dimension attribute (Lines 14-23), level (Lines 25-28) or module (Lines 30-32) should be part of the result.

[8] http://www.w3.org/TR/sparql11-query/#propertypaths

```
1  INSERT {
2   GRAPH ckr:global {
3    ?s ?p ?o
4   }
5  }
6  WHERE {
7   SERVICE <http://localhost:50000/repositories/Base> {
8    SELECT ?s ?p ?o WHERE {
9     GRAPH ckr:global{
10     {
11      ?s ?p ?o.
12     }
13     MINUS{
14      {
15       ?s ?p ?o.
16       ?o rdf:type ?dimAtrVal.
17       ?dimAtrVal rdfs:subClassOf :DimensionAttributeValue.
18      }
19      UNION
20      {
21       ?s rdf:type ?dimAtr.
22       ?dimAtr rdfs:subClassOf :DimensionAttributeValue
23      }
24      UNION
25      {
26       ?s :atLevel ?o.
27       ?o rdf:type :Level
28      }
29      UNION
30      {
31       ?s :hasAssertedModule ?o.
32 }}}}}}
```

Listing 15: Update statement for the insertion of triples from the dimensional model not related to dimension attributes, levels or modules

5.3.1.3 Dimension attributes

The main idea of the update shown in Listing 16 is to copy the dimension attributes which are interesting for the analyst. If the analyst chooses that only data from *Location_Europe* is interesting the dimension attributes which are related to *Location_Asia* do not need to be copied. The query part of the update selects the dimension attributes *d1* and the attributes *d2* they roll up to (Line 13). Furthermore the type *td1* and *td2* of the dimension attributes and the assigned level *l1* and *l2* are selected (Lines 14-19). Lines 20-50 select appropriate dimension attributes. For example, Lines 31-40 select the dimension attributes which roll up to *Location_Europe* and the dimension attributes *Location_Europe* rolls up to. For example, this query may select *d1 Location_Austria* which rolls up to *d2 Location_Europe* with the appropriate levels *l1 Level_Location_Country* and *l2 Level_Location_Continent*.

```
1   INSERT {
2    GRAPH ckr:global{
3     ?d1 :directlyRollsUpTo ?d2.
4     ?d1 :atLevel ?l1.
5     ?d2 :atLevel ?l2.
6     ?d1 rdf:type ?td1.
7     ?d2 rdf:type ?td2.
8    }
9   }
10  WHERE {
11   SERVICE <http://localhost:50000/repositories/Base> {
12    SELECT distinct ?d1 ?td1 ?d2 ?td2 ?l1 ?l2 WHERE {
13     ?d1 :directlyRollsUpTo ?d2.
14     ?d1 :atLevel ?l1.
15     ?d2 :atLevel ?l2.
16     ?d1 rdf:type ?td1.
17     ?td1 rdfs:subClassOf :DimensionAttributeValue.
18     ?d2 rdf:type ?td2.
19     ?td2 rdfs:subClassOf :DimensionAttributeValue.
20     {
21      {
22       ?d1 :directlyRollsUpTo* :Department_All.
23      }
24      UNION
25      {
26       :Department_All :directlyRollsUpTo* ?d1.
27       ?d1 :directlyRollsUpTo* ?d2.
28      }
29     }
30     UNION
31     {
32      {
33       ?d1 :directlyRollsUpTo* :Location_Europe.
34      }
35      UNION
36      {
37       :Location_Europe :directlyRollsUpTo* ?d1.
38       ?d1 :directlyRollsUpTo* ?d2.
39      }
40     }
41     UNION
42     {
43      {
44       ?d1 :directlyRollsUpTo* :Time_All.
45      }
46      UNION
47      {
48       :Time_All :directlyRollsUpTo* ?d1.
49       ?d1 :directlyRollsUpTo* ?d2.
50  }}}}}
```

Listing 16: Update statement for the insertion of dimension attributes related to Department_All, Location_Europe and Time_All with respective levels and types

5.3.1.4 Contexts

The update shown in Listing 17 inserts data generated by the CKR framework. This requires to insert information into the generated graph `infhttp%3A%2F%2Fdkm.fbk.eu%2Fckr%2Fmeta%23global` and the different closure graphs (Lines 1-10). Lines 24-50 query the relevant dimension attributes analogous to the query in Listing 16. Lines 19-21 query the specific contexts *c* which are valid for the dimension attributes *d0 d1 d2*. For those contexts *c* all properties *p* and objects *o* need to be selected (Line 14). Lines 15-16 specify that no objects of contexts *c* asserted by the property *hasAssertedModule* should be selected. This needs to be done by a specific update. As already mentioned every context has a closure defined by a graph. This closures are selected in Lines 17-18 in order to select all triples *cls*, *clp* and *clo* out of the specific *closure*.

```
1   INSERT {
2    GRAPH <springles:infhttp%3A%2F%2Fdkm.fbk.eu%2Fckr%2Fmeta%23global>{
3     ?c ?p ?o.
4     ?c ckr:hasModule ?m.
5     ?closure ckr:closureOf ?c.
6    }
7    GRAPH ?closure{
8     ?cls ?clp ?clo
9    }
10  }
11  WHERE {
12   SERVICE <http://localhost:50000/repositories/Base> {
13    SELECT distinct ?c ?p ?o ?closure ?cls ?clp ?clo ?m WHERE {
14     ?c ?p ?o.
15     FILTER NOT EXISTS
16     {?c :hasAssertedModule ?o.}
17     ?closure ckr:closureOf ?c.
18     GRAPH ?closure {?cls ?clp ?clo.}.
19     ?c :hasDepartment ?d0.
20     ?c :hasLocation ?d1.
21     ?c :hasTime ?d2.
22     OPTIONAL
23     { ?c ckr:hasModule ?m.}
24     {
25      {
26       ?d0 :directlyRollsUpTo* :Department_All.
27      }
28      UNION
29      {
30       :Department_All :directlyRollsUpTo* ?d0.
31      }
32     }
33     {
34      {
35       ?d1 :directlyRollsUpTo* :Location_Europe.
36      }
37      UNION
38      {
39       :Location_Europe :directlyRollsUpTo* ?d1.
40      }
41     }
42     {
43      {
44       ?d2 :directlyRollsUpTo* :Time_All.
45      }
46      UNION
47      {
48       :Time_All :directlyRollsUpTo* ?d2.
49      }
50     }
51    }
52   }
53  }
```

Listing 17: Update statement for the insertion of contexts related to Department_All, Location_Europe and Time_All with respective closure information

5.3.2 Merge

Generally the basic merge operator was implemented by using different updates for the two variants union and intersection which will be explained in the following. Optionally the behavior of the operator may be configured to consider the concept of RDF reification. This behavior was implemented by additional updates which will also be explained. Note that updates regarding RDF reification always need to be executed before the original operators. The example updates assume a configuration of the operator as shown in Listing 8.

5.3.2.1 Union

In contrast to the updates used for the slice/dice operator the updates for this variant of the merge operator needs to be extended by a part to delete specific data. This is because if modules are merged to modules on higher level of abstraction, the lower-level modules need to be deleted. So it is necessary to delete specific triples and graphs. This is done by the *Delete* section of the update shown in Listing 18 in Lines 1-9. From the graphs `infhttp%3A%2F%2Fdkm.fbk.eu%2Fckr%2Fmeta%23global` and `global` the assertion of the contexts and the modules need to be deleted as well as the graph which represent the modules.

Lines 18-30 of Listing 18 basically select all contexts with the respective modules assigned to lower levels than those configured by the analyst. So those modules first need to be merged to a new module and deleted later on. For example, Lines 25-27 select the contexts and modules assigned to the same level *Level_Location_Continent* regarding the property *hasLocation*. At first it needs to be figured out which dimension attributes *r1* are on level *Level_Location_Continent* (Line 27). Then all dimension attributes *d1* with the property *directlyRollsUpTo r1* (Line 26) need to be figured out in order to select the lower-level contexts *ctx*. Note that also those contexts are selected directly asserted to the level *Level_Location_Continent.* One of the last steps shown in Lines 19-21 is the selection of contexts valid for the levels defined by the analyst in order to assert the newly generated module. For the generated module a specific IRI needs to be generated (Lines 31-35). The IRI consists of a *namespace*, a specific description '*UnionModule_'*, so the name differs from original modules, and the concatenated dimension attributes *r0 r1 r2*. Lines 36-39 simply define that the query should not select triples with properties like *rdf:subject, rdf:property, rdf:object* or *:count* which represent reification information. Reification information is handled by separate queries which are described in the following.

```
1  DELETE {
2   GRAPH <springles:infhttp%3A%2F%2Fdkm.fbk.eu%2Fckr%2Fmeta%23global> {
3    ?ctx :hasAssertedModule ?m.
4   }
5   GRAPH ckr:global{
6    ?ctx :hasAssertedModule ?m.
7   }
8   GRAPH ?m {?s ?p ?o}.
9  }
10 INSERT {
11  GRAPH ?u {?s ?p ?o}.
12  GRAPH <springles:infhttp%3A%2F%2Fdkm.fbk.eu%2Fckr%2Fmeta%23global> {
13   ?ctx2 :hasAssertedModule ?u
14  }
15 }
16 WHERE {
17  GRAPH ?m {?s ?p ?o}.
18  ?ctx :hasAssertedModule ?m.
19  ?ctx2 :hasDepartment ?r0.
20  ?ctx2 :hasLocation ?r1.
21  ?ctx2 :hasTime ?r2.
22  ?ctx :hasDepartment ?d0.
23  ?d0 :directlyRollsUpTo* ?r0.
24  ?r0 :atLevel :Level_Department_Department.
25  ?ctx :hasLocation ?d1.
26  ?d1 :directlyRollsUpTo* ?r1.
27  ?r1 :atLevel :Level_Location_Continent.
28  ?ctx :hasTime> ?d2.
29  ?d2 :directlyRollsUpTo* ?r2.
30  ?r2 :atLevel :Level_Time_Year.
31  BIND(IRI(CONCAT('http://dkm.fbk.eu/ckr/olap-model#',
32                  'UnionModule_',
33                  STRAFTER(STR(?r0),'#'),'-',
34                  STRAFTER(STR(?r1),'#'),'-',
35                  STRAFTER(STR(?r2),'#'))) AS ?u).
36  FILTER(?p != rdf:subject).
37  FILTER(?p != rdf:property).
38  FILTER(?p != rdf:object).
39  FILTER(?p != :count).
40 }
```

Listing 18: Update statement for the union variant of the merge operator with levels Level_Department_Department, Level_Location_Continent and Level_Time_Year

The first step to handle reification information is to insert the reifications from the modules into the newly generated module as shown in Listing 19 at Lines 1-9. The functionality of the Lines 29-44 are the same as explained for the update in

```
1  INSERT{
2   GRAPH ?u
3   {
4    [] rdf:subject ?s;
5       rdf:property ?p;
6       rdf:object ?o;
7       :count ?cntSum.
8   }
9  }
10 WHERE{
11  SELECT * WHERE{
12  {
13   SELECT ?s ?p ?o (sum(?cnt) as ?cntSum) ?u WHERE {
14    GRAPH ?m {
15     ?s ?p ?o.
16     FILTER (?p != rdf:subject).
17     FILTER (?p != rdf:property).
18     FILTER (?p != rdf:object).
19     FILTER (?p != :count).
20     FILTER (isLiteral(?o)).
21     OPTIONAL{
22      ?bn rdf:subject ?s;
23          rdf:property ?p;
24          rdf:object ?o;
25          :count ?c.
26     }
27    }.
28    BIND( IF(!BOUND(?c),1,?c) as ?cnt )
29    ?ctx :hasAssertedModule ?m.
30    ?ctx :hasDepartment ?d0.
31    ?d0 :directlyRollsUpTo* ?r0.
32    ?r0 :atLevel :Level_Department_Department.
33    ?ctx :hasLocation ?d1.
34    ?d1 :directlyRollsUpTo* ?r1.
35    ?r1 :atLevel :Level_Location_Continent.
36    ?ctx :hasTime ?d2.
37    ?d2 :directlyRollsUpTo* ?r2.
38    ?r2 :atLevel :Level_Time_Year.
39    BIND(IRI(CONCAT('http://dkm.fbk.eu/ckr/olap-model#',
40                    'UnionModule_',
41                    STRAFTER(STR(?r0),'#'),'-',
42                    STRAFTER(STR(?r1),'#'),'-',
43                    STRAFTER(STR(?r2),'#'))) AS ?u).
44    }
45    GROUP BY ?s ?p ?o ?u
46   }
47   FILTER(?cntSum > 1).
48  }
49 }
```

Listing 19: Update statement to insert reification information for the union variant of the merge operator with levels Level_Department_Department, Lev-

Listing 18. In Lines 15-20 of all triples *s p o* of the modules *m* are selected but only those where *o* is a literal and *p* is not equal to a property which is used by reification. This is necessary because without this filter also existing reification information

would be selected. In Lines 21-26 the resource *c* of the reification information, which counts the existence of specific triples *s p o*, is selected if there exists one. If no *c* exists the resource *cnt* is set to *1* automatically (Line 28). This is needed if a triple does not have any reification information so it needs to be recorded that it exists once. In order to sum up the resource *c* the result of the query needs to be grouped first (Line 45). When the result is grouped by the resources *s p o* it is possible to count the existence of same triples by summing up the resource *cnt* and bind the value to the resource *cntSum*. Now all information is available in order to insert the reification information into the new module *u*. The filter in Line 47 ensures that no reification information is added into the new module for triples which only exist once. This is necessary because of the statement in Line 28 the variable *cnt* may has the value *1*.

In the update shown in Listing 20 the reification information of the modules to be merged has to be deleted. Therefore all triples *s p o* representing reification information of modules *m* need to be selected (Lines 8-14).

The updates regarding reification cannot be joined to a single query because of the grouping. The reification *Insert* statement of Listing 19 groups all triples *s p o* on the level of the newly generated module. The statement of Listing 20 needs the knowledge of the modules which have to be grouped. They operate on different levels so we decided to split the statements. It is also not possible to join the statement of Listing 20 with the union statement even though the union statement deletes the merged modules. This is because the union statement first copies all content of the modules to the newly generated module and then deletes the merged modules. So the reification information which should be deleted would always first be copied to the new module. This is why the Insert/Delete statements which handle reification information need to be executed before the original operators.

```
1  DELETE {
2   GRAPH ?m{
3    ?s ?p ?o.
4   }
5  }
6  WHERE {
7   SELECT ?s ?p ?o ?m WHERE {
8    GRAPH ?m {
9     ?s ?p ?o.
10    FILTER (?p = rdf:subject ||
11            ?p = rdf:property ||
12            ?p = rdf:object ||
13            ?p = :count).
14 }
15  ?ctx :hasAssertedModule ?m.
16  ?ctx2 :hasDepartment ?r0.
17  ?ctx2 :hasLocation ?r1.
18  ?ctx2 :hasTime ?r2.
19  ?ctx hasDepartment ?d0.
20  ?d0 :directlyRollsUpTo* ?r0.
21  ?r0 :atLevel :Level_Department_Department.
22  ?ctx :hasLocation ?d1.
23  ?d1 :directlyRollsUpTo* ?r1.
24  ?r1 :atLevel Level_Location_Continent.
25  ?ctx :hasTime ?d2.
26  ?d2 :directlyRollsUpTo* ?r2.
27  ?r2 :atLevel :Level_Time_Year.
28  }
29 }
```

Listing 20: Update statement to delete reification information for the union variant of the merge operator with levels Level_Department_Department, Level_Location_Continent and Level_Time_Year

5.3.2.2 Intersection

This variant of the merge operator is similar to the union variant but it does not simply merge all triples. It merges only those triples which do exist in every single module. As already mentioned in Section 4.2, the support of RDF reification may not make sense for all kind of scenarios. However, the intersection variant is also able to support RDF reification.

The *Delete* section of the update in Listing 21 is almost equal to the one shown in Listing 18 for the union variant. The only difference is Line 8 where the variables *sdel pdel odel* are used instead of *s p o*. The *Insert* section is equal for both variants union and intersection. If different sections of an update statement are equal to parts of a statement which already got explained the section is marked by the use of three dots as shown in Line 11.

```
1  DELETE {
2   GRAPH <springles:infhttp%3A%2F%2Fdkm.fbk.eu%2Fckr%2Fmeta%23global>{
3    ?ctx :hasAssertedModule ?m.
4   }
5   GRAPH ckr:global {
6    ?ctx :hasAssertedModule ?m.
7   }
8   GRAPH ?m {?sdel ?pdel ?odel}.
9  }
10 INSERT {
11  …
12 }
13 WHERE {
14  SELECT * WHERE {
15   {
16    SELECT ?s ?p ?o ?ctx2 ?u WHERE {
17     {
18      SELECT distinct ?r0 ?r1 ?r2  (count(*) as ?nrOfModules) WHERE {
19       ?ctx :hasAssertedModule ?m.
20       ?ctx :hasDepartment ?d0.
21       ?d0 :directlyRollsUpTo* ?r0.
22       ?r0 :atLevel :Level_Department_All .
23       ?ctx :hasLocation ?d1.
24       ?d1 :directlyRollsUpTo* ?r1.
25       ?r1 :atLevel Level_Location_Continent.
26       ?ctx :hasTime ?d2.
27       ?d2 :directlyRollsUpTo* ?r2.
28       ?r2 :atLevel :Level_Time_All.
29      } GROUP BY ?r0 ?r1 ?r2
30     }
31     {
32      SELECT distinct ?ctx2 ?r0 ?r1 ?r2  ?s ?p ?o
33             (count(*) as ?nrOfEqualTriples) WHERE{
34       GRAPH ?m {?s ?p ?o}.
35       ?ctx2 :hasDepartment ?r0.
36       ?ctx2 :hasLocation ?r1.
37       ?ctx2 :hasTime ?r2.
38       …
39      } GROUP BY ?r0 ?r1 ?r2 ?s ?p ?o ?ctx2
40     }
41     FILTER(?nrOfEqualTriples = ?nrOfModules).
42     FILTER(?p != rdf:subject).
43     FILTER(?p != rdf:property).
44     FILTER(?p != rdf:object).
45     FILTER(?p != :count).
46     BIND(IRI(CONCAT('http://dkm.fbk.eu/ckr/olap-model#',
47                     'IntersectionModule_',
48                      STRAFTER(STR(?r0),'#'),'-',
49                      STRAFTER(STR(?r1),'#'),'-',
50                      STRAFTER(STR(?r2),'#'))) AS ?u).
51    }
52   }
53   {
54    SELECT ?ctx ?m ?sdel ?pdel ?odel WHERE{
55     GRAPH ?m {?sdel ?pdel ?odel}.
56     …
57 }}}}
```

Listing 21: Update statement for the intersection variant of the merge operator with levels Level_Department_All, Level_Location_Continent and Level_Time_All

The update of Listing 21 is divided into different subqueries. Lines 18-29 basically follow the concept of selecting contexts with dimension attributes which roll up to a specific level and was already explained for the union variant. With the grouping operator shown in Line 29 it is possible to group the results by *r0 r1 r2*. Modules are asserted with contexts which do have dimension attributes *d0 d1 d2* that directly roll up to *r0 r1 r2* on specific levels. This means that the variable *nrOfModules* contains the number of modules which are asserted to contexts.

Lines 32-39 select all contexts with dimension attributes which roll up to *r0 r1 r2* similar to Lines 18-29. Furthermore the resources *s p o* of the modules are selected (Line 34). With the grouping operator shown in Line 39 it is possible to count the number of equal triples *s p o* of modules asserted to the context *ctx2* with the asserted dimension attributes *r0 r1 r2* (Lines 35-37) and save it to the variable *nrOfEqualTriples*. The filter in Line 41 defines that only those resources should be in the result where the variable *nrOfEqualTriples* is equal to *nrOfModules*. This means that the resources need to be part of all modules in order to be part of the result.

In Lines 53-57 the triples *sdel pdel odel* of the modules *m* asserted to the context *ctx* are selected. So the assertion of the contexts *ctx* and the modules *m* (Lines 2-7) with their content (Line 8) can be deleted.

The insertion of the reification information is shown in Listing 22. The update statement is very similar to the one shown in Listing 21. Additionally it is necessary to select the count resource *c* of the reification information if it exists (Lines 22-27). If it does not exist the variable is set to *1* and bound to the resource *cnt* (Line 33). This resource is summed up based on the grouping and bound to the value *reif*. So now for all equal triples *s p o* the final number of equal triples is calculated. It is not necessary to delete those resources because the whole module is deleted by the intersection variant later on.

```
INSERT {
 GRAPH ?u {
  [] rdf:subject ?s;
     rdf:property ?p;
     rdf:object ?o;
     :count ?reif.
 }
}
WHERE {
 SELECT ?s ?p ?o ?ctx2 ?u ?reif WHERE {
  {
   SELECT distinct ?r0 ?r1 ?r2  (count(*) as ?nrOfModules) WHERE {
    …
   } GROUP BY ?r0 ?r1 ?r2
  }
  {
   SELECT distinct ?ctx2 ?r0 ?r1 ?r2  ?s ?p ?o
          (count(*) as ?nrOfEqualTriples) (sum(?cnt) as ?reif) WHERE {
    ?ctx :hasAssertedModule ?m.
    GRAPH ?m {
     ?s ?p ?o.
     OPTIONAL{
      ?bn rdf:subject ?s;
          rdf:property ?p;
          rdf:object ?o;
          :count ?c.
     }
    }.
    FILTER(?p != rdf:subject).
    FILTER(?p != rdf:property).
    FILTER(?p != rdf:object).
    FILTER(?p != :count).
    BIND( IF(!BOUND(?c),1,?c) as ?cnt )
    …
   } GROUP BY ?r0 ?r1 ?r2 ?s ?p ?o ?ctx2
  }
  FILTER(?nrOfEqualTriples = ?nrOfModules)
  FILTER(isLiteral(?o))
  BIND(IRI(CONCAT('http://dkm.fbk.eu/ckr/olap-model#',
           'IntersectionModule_',
           STRAFTER(STR(?r0),'#'),'-',
           STRAFTER(STR(?r1),'#'),'-',
           STRAFTER(STR(?r2),'#'))) AS ?u).
 }
}
```

Listing 22: Update statement to insert reification information for the intersection variant of the merge operator with levels Level_Department_All, Level_Location_Continent and Level_Time_All

5.3.3 Abstract By Grouping

The update statements of the different abstracts are very long so the updates are split up in different parts for explanation. For the abstracts, knowledge propagation is a very important part. The core functionality of the abstracts is to check the resources of a module if there exists a specific structure. For example, if a resource is asserted to a specific group it needs to be replaced by the group. Those information may be defined in the same module but it is also possible that the information is inferred by modules on higher level of abstraction. The theoretical concept and the way the CKR framework handles knowledge propagation has already been explained in Section 5.1. The updates shown for this operator assume a configuration as shown in Listing 9.

Listing 23 shows an example query for a specific module in Lines 10-18. This part of the query is used to consider knowledge of a specific module as well as inferred knowledge. The variable *closure* is a closure of a context *c* with an asserted module. Furthermore with the *closure* variable it is possible to find all modules which are directly asserted to the context of the *closure* or modules which are asserted to context on higher levels of abstraction. The CKR framework uses the *derivedFrom* property not only for modules. So in this context it is necessary to get only modules which were asserted to contexts with the *hasAssertedModule* property as shown in Line 14. With those statements it is possible to access all content of the different modules bound to the variable *m* as it is shown in Line 15-17.

The main idea of the query shown in Listing 23 is to select all resources which do have specific properties and objects. For this example query the *grouping property* (Line 16), *selection property* (Line 27) and a *selection resource type* (Line 38) have been configured. Every property needs to be checked in an own subquery. If all the properties would be checked in the same subquery this would mean that all resources and properties need to exist in the same module *m* which may not be true. The filter (Line 42) ensures that the resources with a specific property (Lines 20-30) and type (Lines 31-40) are equal.

```
1   …
2   SELECT DISTINCT ?s ?s_g ?p ?o ?o_g ?s_new ?o_new WHERE{
3    GRAPH :Module8{
4     ?s ?p ?o .
5     FILTER (?p != :grouping)
6    }
7    OPTIONAL{
8     SELECT DISTINCT ?s ?s_g WHERE{
9      {
10      SELECT DISTINCT ?s ?s_g WHERE{
11       ?closure ckr:closureOf      ?c.
12       ?c       :hasAssertedModule :Module8.
13       ?closure ckr:derivedFrom    ?m.
14       ?c2      :hasAssertedModule ?m.
15       GRAPH ?m{
16        ?s :grouping ?s_g.
17       }
18      }
19     }
20     {
21      SELECT DISTINCT ?s ?selRes1 WHERE{
22       ?closure ckr:closureOf      ?c.
23       ?c       :hasAssertedModule :Module8.
24       ?closure ckr:derivedFrom    ?m.
25       ?c2      :hasAssertedModule ?m.
26       GRAPH ?m{
27        ?s :provide ?selRes1.
28       }
29      }
30     }
31     {
32      SELECT DISTINCT ?selRes2 WHERE{
33       ?closure ckr:closureOf      ?c.
34       ?c       :hasAssertedModule :Module8.
35       ?closure ckr:derivedFrom    ?m.
36       ?c2      :hasAssertedModule ?m.
37       GRAPH ?m{
38        ?selRes2 rdf:type :Sale
39       }
40      }
41     }
42     FILTER(?selRes1 = ?selRes2)
43    }
44   }
45   OPTIONAL{...}
46   BIND (IF (BOUND(?s_g), ?s_g, ?s) AS ?s_new)
47   BIND (IF (BOUND(?o_g), ?o_g, ?o) AS ?o_new)
48   FILTER (?p != rdf:subject).
49   FILTER (?p != rdf:property).
50   FILTER (?p != rdf:object).
51   FILTER (?p != :count).
52  }
53  …
```

Listing 23: Select part of the update statement for the abstract by grouping operator with grouping property, selection property and selection resource type set

The subqueries of Listing 23 are defined *Optional* because it is not sure that the queried structure actually exists. Line 45 shows a second *Optional* query which is equal to the one shown in Lines 7-44 but the variables *o* and *o_g* are replaced by *s* and *s_g*. This is because sources or objects may be replaced by the respective *grouping* defined. The resources *s* and *o* (Line 4) are then combined with the resources *s* and *o* queried by the optional subqueries of Line 7 and 45.

The next step is to check if a grouping of a resource exists. This is done by the statement in Lines 46-47. If *s_g* or *o_g* is bound, those values are bound to the resources *s_new* or *o_new*. If they are not bound, *s* or *o* are bound to *s_new* or *o_new*. So the variables *s_new* and *o_new* may be used to insert new data into the module because the value of the variables is either the original value of the resource or the grouped resource. It is easy now to delete all original statements *s p o* and insert the resources *s_new p_new o_new* as shown in Listing 24. The filters of Lines 48-51 in Listing 23 define that reification information should be ignored within this query. The statements shown in Line 3-6 query the resources *s p o* from a specific module. The filter defines that there should be no resources selected where *p* is a *grouping property* (Line 5). This is because otherwise the information of grouped resources is lost. For example if there exists the triple *We grouping Company* the resource *We* would be replaced by *Company*. This should not be done because the information of the grouped resources should remain unchanged.

```
1  DELETE
2  {
3   GRAPH :Module8
4   {
5    ?s ?p ?o.
6   }
7  }
8  INSERT
9  {
10  GRAPH :Module8
11  {
12   ?s_new ?p ?o_new.
13  }
14 }
15 …
```

Listing 24: Delete/Insert part of the update statement for the abstract by grouping operator with grouping property, selection property and selection resource type set

Listing 25 shows the query needed to consider existing RDF reification information. The first subquery (Lines 4-25) checks if reification information for the resources *s p o* already exists. If there exist no reification information the variable *cnt* is set to *1*, otherwise it is set to the value of the resource *c*. Also the grouping needs to be considered which is done by the statement in Lines 18-21 which is equal to the statement explained in Listing 23. The grouping in Line 24 makes it possible to sum up the values of *cnt* in order to calculate the number of grouped resources. It is also possible that there already exist reification information for the grouping resource. So in Lines 26-35 the resource *c* is selected if it exists for *s_g p o*. There is no guarantee that reification information exists for the grouping resource, so Lines 36-43 query if there are already triples with the grouping resource as a subject. If such a triple exists, the variable *cnt* is *1*. Lines 44-45 calculate the total of the sums. The filter in Line 46 defines that only resources should be added to the result where total is greater than *1*. This is because no reification information should be added for a triple which only exists once. Consider that for reification information which are used for literals representing numeric values, it is only necessary to focus on the subjects which need to be rolled up. This is because literals themselves can only be objects and so the queries which handle reification only select *s s_g* and not also *o o_g* as it is done in other queries.

The next step is to update reification information selected by the query of Listing 25. This process is divided into two separate update statements. The first one illustrated in Listing 26 deletes (Lines 1-10) reification information which may exist for the grouped resource *s_g* and inserts (Lines 11-20) the new reification information into the module. The second update statement illustrated in Listing 27 simply deletes all reification information of the original grouped resources. Therefore all resources of the graph are selected (Line 9) where the object is a literal (Line 10). For those resources the subject with its properties and objects (Line 11) which stores the reification information (Lines 12-14) needs to be selected in order to delete the reification information (Lines 1-5).

```
1  …
2  SELECT DISTINCT ?bn ?s_g ?p ?o ?cntMeta ?total WHERE{
3   {
4    SELECT DISTINCT ?s_g ?p ?o (sum(?cnt) as ?cntSum) WHERE{
5     {
6      GRAPH :Module8{
7       ?s ?p ?o.
8       FILTER(isLiteral(?o)).
9       OPTIONAL{
10       ?bn rdf:subject ?s;
11           rdf:property ?p;
12           rdf:object ?o;
13           :count ?c.
14      }
15     }
16     BIND( IF(!BOUND(?c),1,?c) as ?cnt )
17    }
18    {
19    SELECT DISTINCT ?s ?s_g WHERE{
20    …
21    }
22   }
23  }
24  GROUP BY ?s_g ?o ?p
25  }
26  OPTIONAL{
27   SELECT ?bn ?s_g ?p ?o (?c as ?cnt) WHERE{
28    GRAPH :Module8{
29     ?bn rdf:subject ?s_g;
30         rdf:property ?p;
31         rdf:object ?o;
32         :count ?c.
33    }
34   }
35  }
36  OPTIONAL{
37   SELECT ?s_g ?p ?o (count(*) as ?cnt) WHERE{
38    GRAPH :Module8{
39     ?s_g ?p ?o.
40    }
41   }
42   GROUP BY ?s_g ?p ?o
43  }
44  BIND( IF(!BOUND(?cnt),0,?cnt) as ?cntMeta )
45  BIND (?cntSum + ?cntMeta AS ?total)
46  FILTER(?total > 1)
47 }
48 …
```

Listing 25: Select part to query reification information for the abstract by grouping operator with grouping property, selection property and selection resource type set

```
1  DELETE
2  {
3   GRAPH :Module8
4   {
5    ?bn rdf:subject ?s_g;
6        rdf:property ?p;
7        rdf:object ?o;
8        :count ?cntMeta.
9   }
10 }
11 INSERT
12 {
13  GRAPH :Module8
14  {
15   [] rdf:subject ?s_g;
16      rdf:property ?p;
17      rdf:object ?o;
18      :count ?total.
19  }
20 }
21 …
```

Listing 26: Delete/Insert part to update reification information for the abstract by grouping operator with grouping property, selection property and selection resource type set

```
1  DELETE{
2   GRAPH :Module8{
3    ?bn ?x ?y.
4   }
5  }
6  WHERE{
7   SELECT ?bn ?x ?y  WHERE{
8    GRAPH :Module8{
9     ?s ?p ?o.
10    FILTER(isLiteral(?o)).
11    ?bn ?x ?y.
12    ?bn rdf:subject ?s;
13        rdf:property ?p;
14        rdf:object ?o;
15   }
16   {
17    SELECT DISTINCT ?s ?s_g WHERE
18    {
19     …
20    }
21   }
22  }
23 }
24 …
```

Listing 27: Delete part to delete reification information for the abstract by grouping operator with grouping property, selection property and selection resource type set

5.3.4 Abstract Property By Grouping

This operator is an extension of the abstract by grouping operator. Whereas the abstract by grouping operator is only able to specify the resources to be grouped, this operator it is able to specifically define which properties should be updated to the grouped resource. Therefore a direction can be specified to define if incoming or outgoing properties should be updated to the grouped resource.

Basically the update statement of the abstract by grouping operator has been implemented in a way the functionalities described above can be implemented with only a few extensions. The two subqueries of Listing 23 in Lines 7-45 in combination with the statement in Line 4 select *s* and *o* with the respective grouping resources *s_g* and *o_g*. If the resource *s* of a triple *s p o* is replaced by *s_g* this means that all outgoing properties *p* of *s* need to be updated to the new resource *s_g*. So the subquery in Lines 7-45 define outgoing properties whereas the subquery in Line 45 define incoming properties.

If only a specific property should be grouped this needs to be defined additionally. Line 4 of the query shown in Listing 23 needs to be extended by an additional filter in order to select only those properties with the property assigned. As shown in Listing 28 the additional filter of Line 5 need to be defined to select only those resources with the property *subCompanyOf*. This means that the grouping is only done for resources with that specific property.

```
1  …
2  GRAPH :Module8{
3   ?s ?p ?o .
4   FILTER (?p != :grouping)
5   FILTER(?p = :subCompanyOf)
6  }
7  …
```

Listing 28: Extension of the abstract by grouping operator for the abstract property by grouping operator

5.3.5 Abstract Property By Source

As already mentioned, this operator is similar to the abstract by grouping and abstract property by grouping. Because of this, the updates of this operator differ to the other operators only in some parts. Listing 29 shows the operator with the configuration shown in Listing 11. The main difference is that the queries of Listing 29 in Lines 8-40 do not select subjects of triples but objects of triples as *s* and *o*. So it is possible to replace the objects by the respective grouping resources *s_g* and *o_g*.

Due to the fact that this abstract is of type resource-generating a new resource needs to be generated for all grouped resources. The generation of the IRI is too long for visualization so it is described textually. The first component is the namespace of the generated resource which can be set by using the appropriate method of the operator. The second component is the name of the subject whose object are rolled up. This guarantees a higher understandability because the relation of subject and rolled up object is encoded in the name of the newly generated resource. The selection property and the grouped property only are concatenated if they are set explicitly for the operator. The last component is the term *'Generated_'* with an attached Universal Unique ID (UUID). The unique id is generated by the use of the Java class *java.util.UUID*. For the execution of an operator one UUID is generated. If a UUID would be generated for every newly generated resource, this may lead to the situation that resources to be grouped are grouped to resources with different UUIDs.

```
1  …
2  SELECT  ?s ?s_new ?p ?o ?o_new ?o_gen ?s_gen WHERE{
3   GRAPH :Module10{
4    ?s ?p ?o.
5    FILTER (?p != :grouping)
6    FILTER (?p = :stockFlow)
7   }
8   OPTIONAL{
9    SELECT DISTINCT ?s ?s_gen WHERE{
10    {
11     SELECT DISTINCT ?s ?source WHERE{
12      ?closure ckr:closureOf      ?c.
13      ?c :hasAssertedModule :Module10.
14      ?closure ckr:derivedFrom    ?m.
15      ?c2      :hasAssertedModule ?m.
16      ?source ?p ?s.
17      GRAPH ?m{
18       ?source :grouping ?s_g.
19    }}}
20    {
21     SELECT DISTINCT ?source ?s WHERE{
22      ?closure ckr:closureOf      ?c.
23      ?c       :hasAssertedModule :Module10.
24      ?closure ckr:derivedFrom    ?m.
25      ?c2      :hasAssertedModule ?m.
26      GRAPH ?m{
27       ?source :provide ?s.
28    }}}
29    {
30     SELECT DISTINCT ?s WHERE{
31      ?closure ckr:closureOf      ?c.
32      ?c       :hasAssertedModule :Module10.
33      ?closure ckr:derivedFrom    ?m.
34      ?c2      :hasAssertedModule ?m.
35      GRAPH ?m{
36       ?s rdf:type :Sale
37    }}}
38    BIND(…) as ?s_gen)  }
39  }
40  OPTIONAL{ … }
41  BIND (IF (BOUND(?s_gen), ?s_gen, ?s) AS ?s_new).
42  BIND (IF (BOUND(?o_gen), ?o_gen, ?o) AS ?o_new).
43  FILTER (?p != rdf:subject).
44  FILTER (?p != rdf:property).
45  FILTER (?p != rdf:object).
46  FILTER (?p != :count).
47 }
48 …
```

Listing 29: Select part of the update statement for the abstract property by source operator with grouping property, selection property, partition property, grouped property and selection resource type set

Listing 30 illustrates the Delete/Insert section of the query shown in Listing 29. The resources *s p o* are deleted while the newly bound resources *s_new p o_new* are inserted. Line 13-14 show the statements to insert the information about grouped resources by the use of the *partition* property.

```
1  DELETE
2  {
3    GRAPH :Module18
4    {
5      ?s ?p ?o.
6    }
7  }
8  INSERT
9  {
10   GRAPH :Module18
11   {
12    ?s_new ?p ?o_new.
13    ?s :partOf ?s_gen.
14    ?o :partOf ?o_gen.
15   }
16 }
```

Listing 30: Delete/Insert part of the update statement for the abstract property by source operator with grouping property, selection property, partition property, grouped property and selection resource type set

The query to handle reification information is slightly different to the ones of the other operators. Due to the fact that a new unique resource is generated it is not possible that such a resource already exists. So there is no need to update existing reification information first. As shown in Listing 31 the update only inserts reification information. The rest of the query is very similar to the query shown in Listing 25. Also the deletion of the reification information of Listing 32 is similar to Listing 27 so it will not be explained one more time.

```
INSERT {
 GRAPH :Module18{
  [] rdf:subject ?s_generated;
     rdf:property ?p;
     rdf:object ?o;
     :count ?total.
}}
WHERE{
 SELECT DISTINCT ?bn ?s_generated ?p ?o ?cntMeta ?total WHERE{
  {
   SELECT DISTINCT ?source ?p ?o (sum(?cnt) as ?cntSum) WHERE{
    {
     GRAPH :Module18{
      ?s ?p ?o.
      FILTER(isLiteral(?o)).
      OPTIONAL{
       ?bn rdf:subject ?s;
           rdf:property ?p;
           rdf:object ?o;
           :count ?c.
    }}
    BIND( IF(!BOUND(?c),1,?c) as ?cnt )
   }
   {
    SELECT DISTINCT ?s ?source WHERE{
     ?closure ckr:closureOf      ?c.
     ?c       :hasAssertedModule :Module18.
     ?closure ckr:derivedFrom    ?m.
     ?c2      :hasAssertedModule ?m.
     ?source ?p ?s.
     GRAPH ?m{
      ?source :grouping ?s_g.
   }}}
   {
    SELECT DISTINCT ?source ?s WHERE{
     ?closure ckr:closureOf      ?c.
     ?c       :hasAssertedModule :Module18.
     ?closure ckr:derivedFrom    ?m.
     ?c2      :hasAssertedModule ?m.
     GRAPH ?m{
      ?source :provide ?s.
   }}}}
  GROUP BY ?source ?o ?p
  }
  BIND( IF(!BOUND(?cnt),0,?cnt) as ?cntMeta )
  BIND (?cntSum + ?cntMeta AS ?total)
  FILTER(?total > 1)
  FILTER(?p = :stockFlow)
  BIND(…) as ?s_generated)
}}
```

Listing 31: Insert part to update reification information for the abstract property by source operator with grouping property, selection property, partition property, grouped property and selection resource type set

```
1  DELETE{
2   GRAPH :Module18{
3    ?bn ?x ?y.
4   }
5  }
6  WHERE{
7   SELECT ?bn ?x ?y  WHERE{
8    GRAPH :Module18{
9     ?s ?p ?o.
10    FILTER(?p = :stockFlow)
11    FILTER(isLiteral(?o)).
12    ?bn ?x ?y.
13    ?bn rdf:subject ?s;
14        rdf:property ?p;
15        rdf:object ?o;
16   }
17   {
18    SELECT DISTINCT ?s ?source WHERE
19    {
20     …
21    }
22   }
23  }
24 }
```

Listing 32: Delete part to delete reification information for the abstract property by source operator with grouping property, selection property, partition property, grouped property and selection resource type set

5.3.6 Abstract Literal By Source

The main focus of this operator is to execute aggregate functions on literals which represent numeric values. The operator persists of only one update statement. The example update assumes a configuration of the operator as illustrated in Listing 12 and additionally the *selectionResourceType* is set to *Sale*.

The query illustrated in Listing 33 selects literals of triples and executes aggregation functions on them. It also considers reification information without the use of an additional query. The query not considering reification information is the same without the usage of the specific reification parts and will not be explained explicitly. Lines 6-7 define that only triples with the property *revenue* are selected. Line 15 defines that the triples need to have an object of type *literal*. Note that the selection of the resources with a specific property is done with a simple filter (Line 7) and not with a subquery like the other operators do. The reason is because it is assumed that the property which directly asserts the literals to the subject needs to be in the same module. Furthermore the subjects need to be selected which are of a specific type (Lines 19-30). Due to the generation of the reification information by the other operators it is now possible to use this information in order to calculate the right literals. This means it is necessary to find out how often a literal exists for the same subject and property. The number of existences is saved by the object *c* of the *count* property (Line 12). With a simple multiplication of the literal and the number of its existences the total value of the literals is calculated (Line 14). Then the grouping is done on the subject and the property (Line 32) so the SUM of the literals can be calculated. Without the filters defined at Lines 16-17 the objects of the properties *rdf:object* and *count* would be selected because they also represent literals. The next step is to select all information which needs to be deleted from the modules. All reification information of the aggregated literals need to be selected for deletion (Lines 35-41). If the aggregate property, for example, is set to SUM the result of the query only includes the SUM but not the summed up literals. This is why the query needs to select the triples of the summed up literals (Lines 43-47). Listing 34 shows the Delete/Insert part of the update which is similar to the other operators and will not be explained one more time.

```
1  …
2  SELECT * WHERE{
3   {
4    SELECT ?source ?p (SUM(?total) as ?result) WHERE{
5     GRAPH :Module8{
6      ?source ?p ?literal.
7      FILTER(?p = :revenue).
8      OPTIONAL{
9       ?bn rdf:subject ?source;
10          rdf:property ?p;
11          rdf:object ?literal;
12          :count ?c.
13     }
14     BIND(IF(BOUND(?c),?literal * ?c, ?literal) as ?total).
15     FILTER(isLiteral(?literal)).
16     FILTER(?p != rdf:object).
17     FILTER(?p != :count).
18    }
19    FILTER EXISTS{
20     SELECT distinct ?source WHERE{
21      ?closure ckr:closureOf      ?c.
22      ?c :hasAssertedModule :Module8.
23      ?closure ckr:derivedFrom    ?m.
24      ?c2      :hasAssertedModule ?m.
25      ?source ?p ?s.
26      GRAPH ?m{
27       ?source rdf:type :Sale.
28      }
29     }
30    }
31   }
32   GROUP BY ?source ?p
33  }
34  {
35   OPTIONAL{
36    GRAPH :Module8{
37     ?bn rdf:subject ?source;
38         rdf:property ?p;
39         rdf:object ?literal;
40         :count ?c.
41    }
42   }
43   OPTIONAL{
44    GRAPH :Module8{
45     ?source ?p ?all.
46    }
47   }
48  }
49 }
50 …
```

Listing 33: Query part for the abstract literal by source operator with aggregate function, aggregate property and selection resource type set

```
1  DELETE
2  {
3    GRAPH :Module8
4    {
5      ?bn rdf:subject ?source;
6          rdf:property ?p;
7          rdf:object ?literal;
8          :count ?c.
9      ?source ?p ?all.
10   }
11 }INSERT
12 {
13   GRAPH :Module8
14   {
15     ?source ?p ?result.
16   }
17 }
18 …
```

Listing 34: Delete/Insert part for the abstract literal by source operator with aggregate function, aggregate property and selection resource type set

5.4 Testing Environment

The API has been tested using JUnit tests The JUnit tests execute ASK queries which represent the expected result of the base data after the execution of the operators. For the temp repository the Apache Jena Fuseki[9] SPARQL Version 2.0 server was used. In order to be able to run queries implemented by this API it is necessary to configure the dataset of the server. The option *unionDefaultGraph*[10] needs to be set so the server interprets the default graph as the union of all named graphs. This is needed because so it is possible to query the default graph in order to receive data from all named graphs. The option can be set by adding the statement *--set tdb:unionDefaultGraph=true* to the java call of the *fuseki-server.jar* in the script file *fuseki-server.bat*.

[9] http://jena.apache.org/documentation/fuseki2/
[10] https://jena.apache.org/documentation/tdb/configuration.html

6 Summary and Future Work

Data produced by companies need to be represented by numeric values in order to be compatible with traditional OLAP systems. Due to the fact that this is not possible for all kinds of data, OLAP cubes with ontology-valued measures extend the ability of data analysis. The implemented API is a proof-of-concept prototype in order to support OLAP operators on ontology-valued measures. The CKR framework is convenient for the generation of additional information about contexts and knowledge propagation within the base data.

This thesis is not about guidelines for modelling business model ontologies. Existing literature presents business model ontologies like REA [8] in order to model business scenarios. To maximize the benefit of using the operators implemented by the API, modelling guidelines for the measures should be created.

The independence of the API from a particular RDF framework was very important while implementing the API. The operators have been implemented using SPARQL. As some of the update statements for the operators are complex they had to be split into separate updates in order to minimize their execution time. While development we found out that the performance mainly depends on two factors. The first one is the structure of the update. In SPARQL there exist a lot of operators which may be combined in different ways in order to produce equal results. But the performance of the operators varies and so it needs to be considered which operators should be combined. Existing approaches about performance of SPARQL queries [18] may be used to optimize the queries. The second one is the type of SPARQL endpoint used. We noticed differences in execution time when run against different endpoints. More formal performance tests, however, will have to be carried out in order to quantify the differences in execution time between different frameworks.

To enable analysts without programming skills to use the API a graphical user interface (GUI) needs to be developed in order to present an easy-to-use and intuitive way of data analytics. It should be possible to choose existing levels and dimension attributes via the GUI in order to configure the operators of the API. Also a clearly arranged presentation of the results should be implemented by the GUI. The look and feel of the GUI may be inspired by existing solutions like Saiku[11] or any other BI tool.

[11] http://www.meteorite.bi/products/saiku

References

1. Schütz, C., Neumayr, B., Schrefl, M.: Business model ontologies in OLAP Cubes. CAiSE 2013. LNCS 7908, 514–529 (2013).

2. Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and Abstract Syntax, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

3. Brickley, D., Guha, R.V.: RDF Schema 1.1 - W3C Recommendation, http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

4. W3C SPARQL Working Group: SPARQL 1.1 Overview, http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/.

5. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: Proceedings of the 14th international conference on World Wide Web. pp. 613–622. ACM Press (2005).

6. Serafini, L., Homola, M.: Contextualized knowledge repositories for the semantic Web. Web Semant. Sci. Serv. Agents World Wide Web. 12-13, 64–87 (2012).

7. Osterwalder, A.: The Business Model Ontology - A Proposition in a Design Science Approach, (2004).

8. Geerts, G.L., McCarthy, W.E.: An ontological analysis of the economic primitives of the extended-REA enterprise information architecture. Int. J. Account. Inf. Syst. 3, 1–16 (2002).

9. Neumayr, B., Schrefl, M., Linner, K.: Semantic cockpit: An ontology-driven, interactive business intelligence tool for comparative data analysis. ER Work. 2011. LNCS 6999, 55–64 (2011).

10. Neumayr, B., Schuetz, C.G., Schrefl, M.: Towards Ontology-driven RDF Analytics. In: MORE BI 2015-3rd International Workshop on Modeling and Reasoning for Business Intelligence (2015).

11. Chen, C., Yan, X., Zhu, F., Han, J., Yu, P.S.: Graph OLAP: Towards online analytical processing on graphs. In: IEEE International Conference on Data Mining. pp. 103–112 (2008).

12. Abelló, A., Romero, O., Pedersen, T.B., Berlanga, R., Nebot, V., Simitsis, A.: Using Semantic Web Technologies for Exploratory OLAP: A Survey. IEEE Trans. on, Knowl. Data Eng. 27, 571–588 (2015).

13. Golfarelli, M., Maio, D., Rizzi, S.: the Dimensional Fact Model: a Conceptual Model for Data Warehouses. Int. J. Coop. Inf. Syst. 07, 215–247 (1998).

14. Neumayr, B., Schrefl, M., Thalheim, B.: Hetero-Homogeneous hierachies in data warehouses. Proc. 7th Asia-Pacific Conf. Concept. Model. CRPIT 110, 61–70 (2010).

15. Neumayr, B., Schütz, C., Schrefl, M.: Semantic enrichment of OLAP cubes: Multi-dimensional ontologies and their representation in SQL and OWL. OTM 2013. LNCS 8185, 624–641 (2013).

16. Bozzato, L., Serafini, L.: Materialization calculus for contexts in the Semantic Web. In: CEUR Workshop Proceedings. pp. 552–572 (2013).

17. Aalst, W. Van Der: Process Cubes: Slicing, Dicing, Rolling Up and Drilling Down Event Data for Process Mining. AP-BPM 2013. LNBIP 159, 1–22 (2013).

18. Loizou, A., Angles, R., Groth, P.: On the formulation of performant SPARQL queries. Web Semant. Sci. Serv. Agents World Wide Web. 31, 1–26 (2014).