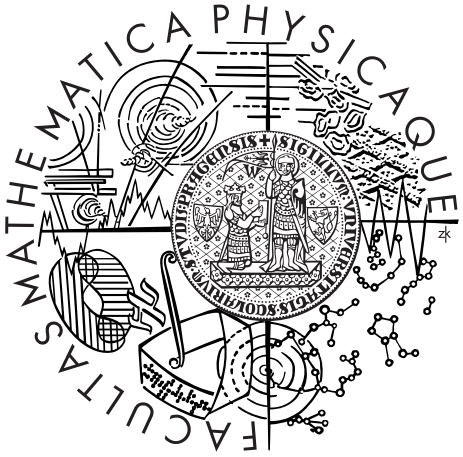
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Miloš Chaloupka

Querying RDF graphs stored in a relational database using SPARQL and R2RML

Department of Software Engineering

Supervisor of the master thesis: Mgr. Martin Nečaský, Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague 2014

I would like to thank everyone who helped me in any way when writing this master thesis. In particular, I thank my supervisor Mgr. Nečaský Martin, Ph.D. for the possibility of detailed consultation on the content of my work.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date

Název práce: Dotazování RDF dat uložených v relační databázi pomocí jazyků SPARQL a R2RML

Autor: Miloš Chaloupka

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Martin Nečaský, Ph.D., Katedra softwarového inženýrství

Abstrakt: RDF formát se stává populárním způsobem jak prezentovat data. Zveřejněná data jsou pak snadno dostupná a i dotazovatelná, bez nějakých podrobných implementačních znalostí. Ale nejčastějším způsobem, jak jsou uložena strukturovaná data, jsou v současné době relační databáze. Ty těží z dlouhé teoretické i praktické historie, nicméně nejsou uzpůsobeny k tomu, aby nějakým snadným způsobem prezentovaly data. Je nezbytné propojit tyto dva světy, pomocí určeného mapování zveřejnit data uložena v relační databázi v RDF formátu. V předložené práci studujeme algebru dotazovacího jazyka SPARQL a vytváříme algoritmus, pomocí kterého jsme schopni vytvořit virtuální SPARQL endpoint nad relačnímy daty. Získané znalosti použijeme k implementaci nástroje, který daný algoritmus používá, čímž ukážeme jeho použitelnost.

Klíčová slova: rdb2rdf, překlad sparql na sql, r2rml, sparql

Title: Querying RDF graphs stored in a relational database using SPARQL and R2RML

Author: Miloš Chaloupka

Department: Department of Software Engineering

Supervisor: Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: The RDF framework is becoming a popular framework for presenting data. It makes the data easily accessible and queryable. But the most common way how to store structured data is to use a relational database systems. The relational databases benefit from their long theoretical and practical history, however the relational database does not offer any convenient way how to publish the data. It is essential to create a mapping between these two worlds, to publish the data stored in a relational database in the RDF format. In the presented work we study the SPARQL algebra and create a transformation algorithm that enable us to create a virtual SPARQL endpoint over the relational data. We apply the acquired knowledge in implementation of a tool which uses the algorithm to proof the concept.

Keywords: rdb2rdf, sparql to sql translation, r2rml, sparql

Contents

Introduction

The aim of this master thesis is to design and implement a SPARQL query processor for RDF data stored in a relational database. The mapping between the relational and RDF representation will be specified with R2RML language. Therefore, it is necessary to analyze the SPARQL query, its structure and prepare an algorithm that will be able to translate the query to the SQL form and then transform the results back to the form as it is expected in the SPARQL query.

Motivation

The RDF framework is becoming a popular framework for presenting data as it is a part of the W3C standard - Linked Data[1]. It makes data easily accessible and queryable without the need to publish information about the data storage, etc. Users are able (using SPARQL queries) to get all information without any further knowledge of a particular implementation. It is only needed to document the used predicates (especially the ones that are specific for the domain) and to describe how to connect to the SPARQL endpoint.

On the other hand, the most common way how to store structured data is to use a relational database systems. The relational databases benefit from their long theoretical and practical history. They usually offer a set of data management services (crash recovery, scalability etc.) and also an optimized relational query processor. So for most structured data there is no intention to store them in some other way than in relational databases. Although there is growing the usage of the non-relational databases they are used mostly for specific cases instead of storing the whole dataset.

However, the relational database does not offer any convenient way how to publish the data. We can open our SQL server (with limited privileges); we can create web services that will work as a gate for the queries. In every case the user needs to know exactly how the data are stored (separation between tables, meanings of the tables and their columns). Also in many cases the user needs to know the exact version of the SQL engine (because they are quite different).

[1]Described at `http://www.w3.org/standards/semanticweb/data` (visited July, 2014)

So we have a data storage and a way how to present the data publicly. The data representation in the relational database and the RDF differs and also the query languages are different so we need somehow close the gap between these two technologies. First task to do is to define how the relational data will be presented, that means the resources and the relationships between them. For that, there are two possible approaches (and both of them are published as a W3C Recommendation). The direct mapping which is a simple transformation of the relational database to the RDF (so it creates the RDF resources according to the table names, their columns and others information like keys, foreign keys, etc.). The other approach is more complex than the direct mapping. It is using a mapping definition (typically in the form of a file) that is manually created and that defines the exact way how to map the relational data into the RDF form.

We can finish now; we have described how the relational data should be published, and there are several tools that allow us to dump the relational database into the RDF form. Moreover, then we can use existing tools to create SPARQL endpoint over an RDF dataset. However, that is not efficient, it will be needed to create a large dump on every change and load it into memory.

However, it is not needed to have all the data dumped from the relational database. We can create a virtual SPARQL endpoint that can be queried without actually storing any RDF data. It only holds the information how to represent the relational data and every SPARQL query converts into an SQL query, executes it against the database and the result is transformed back to the form that is expected from the SPARQL query. Thanks for that we can use the efficient SQL engine to query (although in some cases, it is nearly impossible to create an efficient query) and there is no need to run any workflow when the data changes.

Contribution

To summarize, our main contributions are:

- We analyze the SPARQL algebra and R2RML mapping options
- Using this algebra we propose an algorithm for transformations between SPARQL query and the SQL query
- We propose the possible optimizations

- We build a tool that can be used as a virtual SPARQL endpoint over the relational data

The document structure

The first chapter briefly introduces the key technologies that are used, there is described the difference between the data representation in relational databases and RDF datasets.

In the second chapter there are named some of the works that are somehow related, especially the works that solve some similar task.

The formal definition of the SPARQL algebra can be found in the third chapter. The allowed operations in SPARQL algebra are also described here.

The fourth chapter introduces the transformation algorithm. There is described the exact way how to transform the SPARQL query to the SQL query and how to handle the result from the relational database engine.

In the fifth chapter, there are described optimization options. The optimization that can be done in the SPARQL algebra, in the SQL query, but also other ways to optimize the process.

The sixth chapter contains the evaluation results of the proposed algorithm. The correctness and also the performance of selected queries.

The seventh chapter describes the implementation of the algorithm in the proposed tool.

In the eighth chapter, there is a user guide that will help the user to install, configure and use the implemented tool.

1. Technical background

In this chapter we will describe the key technologies for this work. Moreover, we will mention several terms specific for the technologies that are later used in the work.

1.1 Relational databases

The relational database (see [10]) model is based on the branches of mathematics called the set theory and the predicate logic. An essential element of a relational database is a table (alias relation). The table has its name and a set of columns (that consists from a name and datatype). The data are stored as a row in such table.

The relational model also reflects the dependencies of the columns. We say that the set of columns is a superkey if it is true that for every row in the table there is a unique set of values in the superkey columns. So if we know the values of the superkey we can identify the row that corresponds to it. If there is no subset of the superkey, which is also a superkey, then it is the key of the table.

The modern database systems (RDBMS) also enable us to create constraints on values. These constraints are checked when there is any change of data, and they are used to ensure the data consistency. One of the possible constraints is so called foreign key. It marks a column as a reference to another tables key. So it makes checking whether the referenced value really exists (and also do not allow to delete a row that is referenced from somewhere). It is also representing the information that the two tables have some relationship, that there is some dependency between the tables.

The sample database schema in the figure 1.1 represents a storage for NUTS and LAU regions. Regions of both types are identified by their code, and there are also references between them. The LAU regions with level two have defined their parent region which has level one. The LAU regions with level 1 do not have a parent region from LAU, but they do have a reference to a parent NUTS region. The NUTS regions do have a reference to its parent NUTS region which has a lower level, only the NUTS regions with level 0 do not have any parent. The same data can also be stored in other ways. They can be separated into multiple

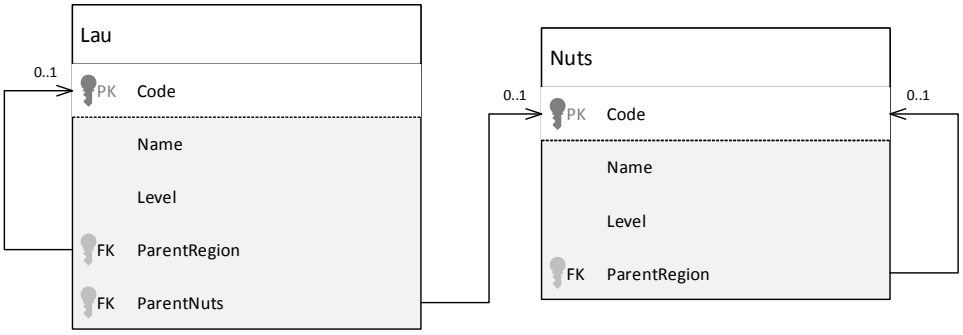


Figure 1.1: Database schema sample

tables (4 tables for NUTS regions with levels 0-3 and 2 tables for LAU regions with levels 1 and 2). Alternatively, the other way they can be merged into one table (with an indicator whether it is a NUTS or an LAU region). Sample data that can be stored in a database with a schema from the figure 1.1 are shown in the figure 1.2.

Lau				
Code	**Name**	**Level**	**ParentRegion**	**ParentNuts**
CZ0201	Benešov	1	*NULL*	CZ020
CZ0202	Beroun	1	*NULL*	CZ020
CZ0203	Kladno	1	*NULL*	CZ020
529303	Benešov	2	CZ0201	*NULL*
529516	Čerčany	2	CZ0201	*NULL*
529451	Bystřice	2	CZ0201	*NULL*
531791	Svatá	2	CZ0202	*NULL*

Nuts			
Code	**Name**	**Level**	**ParentRegion**
CZ	ČESKÁ REPUBLIKA	0	*NULL*
CZ0	ČESKÁ REPUBLIKA	1	CZ
CZ01	Praha	2	CZ0
CZ02	Střední Čechy	2	CZ0
CZ010	Hlavní město Praha	3	CZ01
CZ020	Středočeský kraj	3	CZ02

Figure 1.2: Sample relational data

During the years, it has become an industry standard, and it is nowadays the most common way how to store the structured data. Although there are appearing lots of new technologies that handle the data storage (non-relational databases) and although the big companies (like Google, Facebook, Twitter, etc.) are starting to use them (but never completely replace the RDBMS), most of the

companies store the data using the RDBMS.

1.1.1 SQL

When we have a relational database, we need a language to work with it. It is called the structured query language (SQL). It offers the ability to get or update data in the database, but when we want to make a query, we need to know the exact database structure. So when we have to query the same data in two different systems we will make most likely two different queries.

The SQL language is standardized however different RDBMSs have various syntax for several most advanced queries (like recursive queries), and some queries are specific only for selected RDBMS implementation. So for many queries we need also to know which type and version of RDBMS is used. In this work, we use the T-SQL syntax (see [9]) that works with the MS SQL.

```
1 SELECT L2.Name AS Lau2, N3.Name AS Nuts3
2   FROM Lau AS L2
3   INNER JOIN Lau AS L1 ON L2.ParentRegion = L1.Code
4   INNER JOIN Nuts AS N3 ON L1.ParentNuts = N3.Code
```

Figure 1.3: Sample SQL query

#	Lau2	Nuts3
1	Benešov	Středočeský kraj
2	Čerčany	Středočeský kraj
3	Bystřice	Středočeský kraj
4	Svatá	Středočeský kraj

Figure 1.4: Sample SQL query result

The sample SQL query from the figure 1.3 for the database with schema from the figure 1.1 selects the LAU2 regions names also including the names of their corresponding NUTS3 region names. When creating the query we need to know how are the data exactly stored. The way how the data are stored also can contain additional information. For example in the sample query, we do not ask for the level of Lau region, because we know, that the Lau regions which do have a parent Lau region must have their level set to 2.

When we execute the SQL query from the figure 1.3 against the relational database from the figure 1.2, we will get the result shown in the figure 1.4.

1.2 RDF

The resource description framework (RDF, see [1]) is a framework for representing data. The data are understood as directed graphs where values are stored in nodes with named oriented edges between them. In other words, the graph is a set of subject-predicate-object triples that represent the fact that the node subject is connected to the node object using the edge named predicate.

The values in RDF (so-called RDF terms) have three different types: IRIs (Internationalized resource identifier), literals and blank nodes. The literals and IRIs denote resources (entities). The resource denoted by an IRI is called its referent, and the IRIs have global scope, so the same IRI always denotes the same resource. We can understand it as an identifier, and it does not matter whether we find the IRI in subject or object, it always represents the same resource. The blank nodes also represent a resource, but not a particular resource. It only creates the statement that there is something with the given relationships, but it does not have any particular identifier.

Therefore, the subject can be only an IRI or a blank node, but the object can be an IRI, a blank node and also a literal. The relationships between nodes are always specific so the predicate can be only an IRI.

The sample data from the figure 1.5 contain information about several NUTS region. They contain the same information as the table Nuts from the figure 1.2. About every NUTS region the data state its name, code, references to its parent and sub-regions and the region is identified by a unique IRI (containing the region code). There are several ways how to write the RDF data, in the sample there is used the Turtle language (see [4]).

```
@prefix ec: <http://ec.europa.eu/eurostat/ramon/ontologies/geographic.rdf#> .

<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ010> a ec:NUTSRegion;
 ec:level 3;
 ec:name "Hlavní město Praha";
 ec:regionCode "CZ010";
 ec:hasParentRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ01>.

<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ020> a ec:NUTSRegion;
 ec:level 3;
 ec:name "Středočeský kraj";
 ec:regionCode "CZ020";
 ec:hasParentRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ02>.

<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ01> a ec:NUTSRegion;
 ec:level 2;
 ec:name "Praha";
 ec:regionCode "CZ01";
 ec:hasParentRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ0>;
 ec:hasSubRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ010>.

<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ02> a ec:NUTSRegion;
 ec:level 2;
 ec:name "Střední Čechy";
 ec:regionCode "CZ02";
 ec:hasParentRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ0>;
 ec:hasSubRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ020>.

<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ0> a ec:NUTSRegion;
 ec:level 1;
 ec:name "ČESKÁ REPUBLIKA";
 ec:regionCode "CZ0";
 ec:hasParentRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ>;
 ec:hasSubRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ01>;
 ec:hasSubRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ02>.

<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ> a ec:NUTSRegion;
 ec:level 0;
 ec:name "ČESKÁ REPUBLIKA";
 ec:regionCode "CZ";
 ec:hasSubRegion <http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ0>.
```

Figure 1.5: Sample RDF data

1.2.1 SPARQL

The SPARQL (SPARQL Protocol and RDF Query Language, see [2]) language is created to query the RDF sources. A significant advantage of this language is that we do not need to know the way, how the data are stored in the queried source. We only need to know what kind of entities are present in the source and what properties we can find on them. However, that can be also retrieved by a query so in fact we are able to query the source with no more knowledge than how to connect to the endpoint.

Although the SPARQL language also offers the possibility to update data (in the version 1.1), we will discuss only the data retrieval. However, also for

that it provides us several options how to query, and it can differ in the result type - it can be a SPARQL result set (from a SELECT statement), graph (from a CONSTRUCT or a DESCRIBE statement) or a boolean value (from ASK statement). Also many endpoints allow us to select the datatype of the result (whether it should be an XML, CSV and so on).

The SPARQL language is specially designed for the RDF data sources, so it is (as expected) build from triple patterns (selecting triples from the dataset and the terms from it) and several other language constructs (that are somehow similar to the SQL language).

```
1  PREFIX ec: <http://ec.europa.eu/eurostat/ramon/ontologies/
       ↪ geographic.rdf#>
2
3  SELECT ?name1 ?name3
4  WHERE
5  {
6    ?nuts1 ec:level 1 ;
7      ec:hasSubRegion/ec:hasSubRegion ?nuts3 ;
8      ec:name ?name1 .
9    ?nuts3 ec:name ?name3 .
10 }
```

Figure 1.6: Sample SPARQL Query

#	**name1**	**name3**
1	ČESKÁ REPUBLIKA	Hlavní město Praha
2	ČESKÁ REPUBLIKA	Středočeský kraj

Figure 1.7: Sample SPARQL query result

The sample SPARQL query from the figure 1.6 gets the names of the NUTS 1 regions and the names of the subregions of their subregions (that means NUTS 3 names that are descendant from the NUTS 1 region). The query result (when the query is executed against the dataset from figure 1.5) get the result shown in the figure 1.7.

1.3 R2RML

As we mentioned the relational databases are suited to store the structured data but the RDF framework is better to publish the data. However, the data

representation is different, so we need somehow close the gap between these two approaches. The R2RML language (see [3]) is a language to express the mapping from a relational database to an RDF dataset. It allows us to present the data stored in a relational database as an RDF dataset. There are more languages and methods for this kind of mapping, but the R2RML language is a standard proposed by the World Wide Web Consortium (W3C).

The R2RML offers the option to define completely customized view over the relational data. In the mapping definition, we list triples definition that contains information how to query the relational database and how to generate the RDF triples from the returned rows. The generation of the RDF triple can be straightforward (from a column value or a constant), or the triple can be generated using a template (possibly combining several column values).

The R2RML mapping can be used by some tool for example to dump the data or as this work proposes, to offer a virtual SPARQL endpoint.

The sample mapping shown in the figure 1.8 maps the table Nuts from the database schema from the figure 1.1. As it can be seen the R2RML mapping definition is an RDF file. So it is possible to parse and process the mapping using the standard RDF tools. This example is written using the Turtle language.

To create the RDF data using the mapping from the figure 1.8, we proceed as follows for every triples map (in the sample there is only a single triples map `<Nuts>`). The first step is to retrieve the data from the database. The query is defined in the `rr:logicalTable` node, in this sample there is written a table that contains the data but it is possible to write any custom SQL query.

The next step is to convert the returned result into the RDF triples. The same algorithm is repeated for every row. We show the process on the first row from the table Nuts in the figure 1.1 namely the row with the following values: `Code` = CZ, `Name` = ČESKÁ REPUBLIKA, `Level` = 0 and `ParentRegion` = NULL.

The subject of all triples generated from the selected row is the same, and it is defined in the `rr:subjectMap` node. In this sample, it is defined using a template where the column references will be replaced by the proper values. The application of the pattern on the selected row creates the subject `http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ`.

```
@prefix ec: <http://ec.europa.eu/eurostat/ramon/ontologies/geographic.rdf#> .

<Nuts> a rr:TriplesMap;
 rr:logicalTable [ rr:tableName "[dbo].[Nuts]"; ];
 rr:subjectMap [
  rr:template "http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/{Code}";
  rr:class ec:NUTSRegion
 ];
 rr:predicateObjectMap [
  rr:predicate ec:level;
  rr:objectMap [
   rr:column "Level";
   rr:datatype xsd:int;
  ];
 ];
 rr:predicateObjectMap [
  rr:predicate ec:name;
  rr:objectMap [ rr:column "[Name]"; ];
 ];
 rr:predicateObjectMap [
  rr:predicate ec:regionCode;
  rr:objectMap [ rr:column "[Code]"; ];
 ];
 rr:predicateObjectMap [
  rr:predicate ec:hasSubRegion;
  rr:objectMap [
   rr:parentTriplesMap <Nuts>;
   rr:joinCondition [
     rr:child "[Code]";
     rr:parent "[ParentRegion]";
   ];
  ];
 ];
 rr:predicateObjectMap [
  rr:predicate ec:hasParentRegion;
  rr:objectMap [
   rr:parentTriplesMap <Nuts>;
   rr:joinCondition [
    rr:child "[ParentRegion]";
    rr:parent "[Code]";
   ];
  ];
 ];
.
```

Figure 1.8: Sample R2RML mapping

From the returned row, we create only triples defined by the mappings that do not contain a reference to a triple map (even if it is a reference to the same triples map). The mappings that do contain the reference are handled separately (in this sample the triples with the predicates `ec:hasSubRegion` and `ec:hasParentRegion`).

If the subject map contains the `rr:class` node, we use it to create the first RDF triple. The predicate is `rdf:type` and the value is defined in the `rr:class` node. In this sample there is a constant `ec:NUTSRegion`, so the created triple is:

- `<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ>` `rdf:type ec:NUTSRegion`

The node `rr:class` may also contain more complex definition than a constant, and then it is written in the same way as the `rr:objectMap` node (with same possibilities).

After that, we process every `rr:objectMap` node (only the ones that do not contain a reference to a triple map). Their parent node (`rr:predicateObjectMap` contains the definition of the predicate (in this sample the predicates are constants and that is probably the most common way but it is possible to define the predicates using the values from the row) and the definition of the object. In this sample, the object maps have their value created from the specified column value. For the selected row we will create following triples:

- `<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ>` `ec:level 0`
- `<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ>` `ec:name "ČESKÁ REPUBLIKA"`
- `<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ>` `ec:regionCode "CZ"`

Using the same algorithm we process every returned row from the database. The only missing step is to process the object maps that do contain a reference to a triple map. They are processed separately. We will show the algorithm for the object map that is used with the `ec:hasSubRegion` predicate. The referencing object map needs a special SQL query to generate the triples. It has to join two queries, to generate both subjects that will be connected with the predicate. The join condition is defined in the `rr:joinCondition` node. For this referencing object map we will get the query that is in the figure 1.9.

```
1 SELECT Child.Code, Parent.Code
2   FROM Nuts AS Child
3   INNER JOIN Nuts AS Parent ON Child.Code = Parent.ParentRegion
```

Figure 1.9: SQL query for the referencing object map

The triples are created from the returned rows by the application of the subject maps of the used triple maps. The child triple map (the one where the object map is present) uses the columns from the `Child` source, and the parent triple map (the one that is referenced) uses the columns from the `Parent` source. So,

for example, the returned row with values `Child.Code` = ”CZ” and `Parent.Code` = ”CZO” results in the following triple:

- `<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ>`
  `ec:hasSubRegion`
  `<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/CZ0>`

When the sample mapping is applied on a database with data from the figure 1.2, it will result in the triples shown in the RDF sample (the figure 1.5).

2. Related work

In this chapter, we will name several works that are related with this paper or the implementation. First we will mention the tools that aim the similar task as our work. Also, then we will also mention the tools that are used for our implementation (dotNetRDF and r2rml4net) and a tool that should be able to use our implementation as a data source (Payola).

2.1 Virtuoso Universal Server

The Virtuoso Universal Server[1] is a multi-model data server. It offers the relational, XML, RDF and free text storage and several services with their proper engines. It also offers several services to access these engines. The SPARQL endpoint, ODBC connectors, document web server and so on.

The product is available in two different licenses. As a commercial product and as an open source project (under GNU GPL v2 license).

In term of RDF storage, it works as a native storage. We fill it with triples and then we are able to use the SPARQL endpoint over these loaded data. It is also possible to define the Linked Data View over the RDBMS data source. That is exactly what we are doing, but they are using their own mapping language, and it is not possible to use that engine over other RDBMS.

Although it is possible to convert the R2RML file into their own mapping language, it is quite limiting that it is not possible to use the mapping engine over another RDBMS because the usage of the Virtuoso Universal Server as the relational database is very minor in compare with the ORACLE, MSSQL, MySQL and others.

For our purposes, this is an excellent tool to compare the correctness and performance with our implementation. However, we believe that for most of the companies it will be more convenient to use an extra mapping tool over their running RDBMS than buying a new one and migrating all data (and possibly modifying the queries if there will be some differences).

[1] Available at `http://virtuoso.openlinksw.com/` (visited June, 2014)

2.2 Morph

Morph[2] is a tool created by Jean-Paul Calbimonte that uses the R2RML mapping file to generate a dump of RDF triples from a relational database. It is an open source software under Apache License v2.

It is written in Scala language (running on the Java Virtual Machine) and the functional language seems to fit the task nicely. It is still in question whether it will not be good to implement a part of our tool using F# (a functional language in the .NET world) because the work with the mappings and the algebra may be more efficient and more transparent using the functional approach.

During the creation of this paper there was also created Morph-RDB[3] project (under the Ontology Engineering Group), that added support for the SPARQL language and now they are focusing the integration with the Linked Data Platform. They state the support for the MySQL, PostgreSQL and MonetDB databases.

2.3 TARQL

TARQL[4] is a tool created by Richard Cyganiak to query CSV files using the SPARQL syntax.

TARQL is a command line tool, and everything is passed as a parameter. Several options that allow us to develop the SPARQL query correctly, forcing some behaviour and so on. Also, the parameter where is defined the file with the actual SPARQL query.

It enables to run a SPARQL query where we define a CSV file in the `FROM` statement (or by passing a parameter to the tool). It is equivalent to executing the query with the `VALUES` filled by the content of the CSV file. Moreover, it can detect the column headings (if there is some), so then the variables are named according to the headings.

This tool is very useful when we get a data dump in a CSV file, and we want

[2]Git repository available at `https://github.com/jpcik/morph` (visited June, 2014)
[3]Git repository available at `https://github.com/oeg-upm/morph-rdb` (visited June, 2014)
[4]Git repository available at `https://github.com/cygri/tarql` (visited June, 2014)

to generate the RDF triples. Using the `CONSTRUCT` query, we can achieve this very quickly using the standard language we already know.

2.4 D2RQ Platform

The D2RQ Platform[5] is a set of tools (created by Richard Cyganiak just as TARQL) that can be used to map relational database to RDF data. It can generate the mapping file and then to dump the data in RDF format or to create a virtual SPARQL endpoint over the mapped relational data.

The D2RQ Platform is written in Java and supports several databases (like Oracle, MySQL, PostgreSQL and several others). It is also possible to use the platform as part of other Java applications, using the Jena API[6].

It uses its own mapping language (or it is possible to use the direct mapping mechanism), but they are planning to support the R2RML language. However, currently the support is only in the development branch. Even after several attempts we were not able to run complex queries using our sample data and environment.

The D2RQ Platform is a state-of-the-art for the mapping from relational databases to the RDF data. The D2RQ mapping language was used to develop the R2RML language (Richard Cyganiak is a joint author of the specification).

2.5 dotNetRDF

The dotNetRDF[7] is the major .NET library for the semantic web applications. It offers an API to work with RDF, SPARQL and the semantic web.

As part of the dotNetRDF library, there are also RDF store (in-memory) with their own SPARQL engine called Leviathan. If we do not want to use the in-memory store, we need to use some third party store that has implemented a connector in the DotNetRDF library. There are available several connectors to

[5] Available at `http://d2rq.org/` (visited June, 2014)

[6] Apache Jena is an opensource framework for building semantic web applications, available at `https://jena.apache.org/` (visited June, 2014)

[7] Available at `http://www.dotnetrdf.org/` (visited June, 2014)

the native RDF stores, for example for the Virtuoso Universal Server.

The dotNetRDF had developed an SQL backend as an RDF store. They store the triples directly in the relational database. They moved away because it has not proven as a performant store.

The library also offers API to develop custom storages, including the ability to parse SPARQL language into their algebra representation (the algebra is optimized for the Leviathan, but it is very similar to the official SPARQL algebra).

2.6 r2rml4net

The r2rml4net[8] is a library for reading and processing the R2RML mapping files.

The library provides functions to load mapping from file, create it from code or generate it from the database schema (creates R2RML mapping analogous to the direct mapping).

There are also functions to generate the RDF dump from the relational database. However, there is no support for the virtual SPARQL endpoint. That is the gap that will be filled by the proposed tool.

2.7 Payola

Payola[9] is a web application which offers visualization for the RDF data. It includes several visualization plugins for the RDF data, and it can get the data from a SPARQL endpoint.

The Payola works with analysis. The analysis is an algorithm how to process the data. It contains the information how to get the data (it can merge data from sources like SPARQL endpoint) and how to process the gained data into the final form. Then it is possible to visualize the final RDF data created in the analysis.

One of the analysis (COI.CZ inspections and sanctions by regions and sanction

[8] Git repository available at `https://bitbucket.org/r2rml4net/core` (visited June, 2014)
[9] Available at `http://payola.cz` (visited June, 2014)

value[10]) was the aim of our implementation. The implementation supports the SPARQL queries that are needed for this analysis.

[10] Available at `http://live.payola.cz/analysis/7b2ee8cc-f03a-4a04-ba9d-e54e65346191` (visited June, 2014)

3. SPARQL algebra

In this chapter we introduce the SPARQL algebra, the needed properties of its parts and also the possible transformation of a query. There is an official algebra described in [2], and the following definitions are inspired by this official algebra. However, we propose the definitions in a way, that we can analyze it and process it to the corresponding SQL query. There is also another algebra for SPARQL, proposed in [6] exactly for converting to SQL, but it does not cover all possibilities in SPARQL and therefore it does not solve all issues.

3.1 Query parts

First we need to define the elements of the SPARQL algebra. Basic elements are an RDF term and a Query variable. The RDF term can be an IRI, RDF Literal or a blank node. The Query variable is a name, bound to some RDF term during the processing of the result of SPARQL query. These elements are defined in the SPARQL Query Language [2].

Definition 3.1. Let *RDF-I* be the set of all IRIs, *RDF-L* be the set of all literals and *RDF-B* be the set of all blank nodes.
Then the set of **RDF terms** is defined as $RDF\text{-}T = RDF\text{-}I \cup RDF\text{-}L \cup RDF\text{-}B$. □

The SPARQL Query Language document [2] introduces a solution mapping. The solution mapping is a mapping from a set of query variables to a set of RDF terms. The result of any SPARQL query can be seen as a sequence of solution mappings.

Definition 3.2. A **solution mapping** μ is a partial function, $\mu : V \to RDF\text{-}T$, where V is the set of variables and *RDF-T* is the set of RDF terms. The domain of μ (the subset of V where μ is defined) we denote $dom(\mu)$.
A **solution sequence** M is a list of solution mappings, $M = \{\mu_1, \mu_2, ...\}$. The domain of the solution sequence is $dom(M) = \bigcup_{\mu \in M} dom(\mu)$. □

Note 3.2.1. Also any RDF triple can be understood as a solution mapping. The domain will contain three variables representing subject, predicate and object in the RDF triple. For example $dom(\mu) = \{?\texttt{subject}, ?\texttt{predicate}, ?\texttt{object}\}$.

These variables will be mapped into the corresponding RDF terms of the RDF triple. ∘

Definition 3.3. To simplify the following definitions, we introduce also following operators on solution mappings:

- **Join**: $\mu = \mu_1 \bowtie \mu_2$ where μ, μ_1 and μ_2 are solution mappings. It means that $dom(\mu) = dom(\mu_1) \cup dom(\mu_2), \mu(dom(\mu_1)) = \mu_1(dom(\mu_1)), \mu(dom(\mu_2)) = \mu_2(dom(\mu_2))$. The $\bowtie$ operator is defined only if $\mu_1(dom(\mu_1) \cap dom(\mu_2)) = \mu_2(dom(\mu_1) \cap dom(\mu_2))$. We mark $\mu_1 \sim \mu_2$ when $\mu_1 \bowtie \mu_2$ is defined (μ_1 and μ_2 are compatible), $\mu_1 \nsim \mu_2$ otherwise (μ_1 and μ_2 are incompatible).
- **Reduce**: $\mu = \rho(\mu', V_x)$ where μ and μ' are solution mappings and $V_x \subseteq V$ is a set of variables. It means that $dom(\mu) = dom(\mu') \cap V_x$ and $\mu(dom(\mu)) = \mu'(dom(\mu))$.
- **Equal**: We mark $\mu_1 \equiv \mu_2$ when $dom(\mu_1) = dom(\mu_2)$ and $\mu_1 \sim \mu_2$ otherwise we mark $\mu_1 \not\equiv \mu_2$.

□

Note 3.3.1. The $\bowtie$ operator joins two solution mappings into one resulting, that has mapped the variables into same values as source solution mappings - so the shared variables must be mapped to the same values.
The ρ operator reduces the source solution mapping into the one that has smaller domain, but binds the variables to the same values as the source one.
We say that two solution mappings are equal if they have the same domain and they are compatible. ∘

Note 3.3.2. Sample usage of the defined operators is in the figure 3.1. ∘

The solution sequence is a result of a query over a source dataset. So we introduce also a formal definition what is a dataset and what is a query.

Definition 3.4. The **RDF Dataset** is the set of the default graph and the named graphs (with their names). $DS = \{G, (u_1, G_1), (u_2, G_2), ..., (u_n, G_n)\}$ where G and each G_i are graphs and each u_i is an unique IRI. The G is called default graph. □

Definition 3.5. The **SPARQL Query** is a mapping Q: $DS \to \mathcal{P}(M_Q)$ where DS is the source dataset, and $\mathcal{P}(M_Q)$ is the set of all possible solution sequences with variables (possibly) bound in Q. The domain of query Q is equal to the set of (possibly) bound variables in resulting solution sequences and it is marked

$$\mu_1 = \{v_1 \to a, v_2 \to b\}$$
$$\mu_2 = \{v_1 \to a, v_3 \to c\}$$
$$\mu_3 = \{v_1 \to a, v_2 \to c\}$$
$$\mu_4 = \{v_3 \to c\}$$

(a) Solution mappings

$$\mu_1 \bowtie \mu_2 = \{v_1 \to a, v_2 \to b, v_3 \to c\}$$
$$\mu_1 \bowtie \mu_3 \text{ is not defined}$$
$$\mu_1 \bowtie \mu_4 = \{v_1 \to a, v_2 \to b, v_3 \to c\}$$

(b) The $\bowtie$ (join) operator

$$\rho(\mu_1, \{v_1\}) = \{v_1 \to a\}$$
$$\rho(\mu_2, \{v_2\}) = \{\}$$
$$\rho(\mu_3, \{v_1, v_3\}) = \{v_1 \to a\}$$

(c) The ρ (reduce) operator

$$\mu_1 \not\equiv \mu_2$$
$$\mu_1 \equiv \mu_2$$
$$(\mu_1 \bowtie \mu_2) \equiv (\mu_1 \bowtie \mu_4)$$
$$\mu_1 \not\equiv \mu_4$$

(d) The $\equiv$ (equal) operator

Figure 3.1: Samples for the solution mapping operators

$dom(Q)$. The **incomplete** SPARQL Query is a mapping $Q\colon DS \times \mathcal{P}(V_e) \to \mathcal{P}(M)$ where V_e is the set of external variables, denoted $dom^e(Q)$. The external variable is a variable that is used in the query, and moreover, it may be unbound. $\mathcal{P}(V_e)$ is the set of all possible solution mappings which domains are a subset or equal to V_e. $\mathcal{P}(M_Q)$ is the set of all possible solution sequences with variables (possibly) bound in Q. If a variable is an external variable but it is not in the domain of the query (that means that the variable is used, but it is never bound), we say that the variable is out of the scope. The set of the out of scope variables we denote as the out of the scope domain, marked $dom^o(Q)$. □

Note 3.5.1. The out of the scope variable is a variable that is used in the query, but the query does not introduce a value for the variable. The following definitions include the information how the set is defined or modified using the defined operators. ∘

Note 3.5.2. All variables out of the scope are used in the query and are unbound, so $dom^o(Q) \subseteq dom^e(Q)$. According to the definition, we can determine the out of the scope domain, $dom^o(Q) = dom^e(Q) \setminus dom(Q)$. ∘

Note 3.5.3. The external variables may influence the evaluation using a value that is coming from another source than the query itself. If we specify the value of the external variable and it is also bound in the query, the value must be the same. So for $Q(DS, \mu^e) \to M$ is true that $\forall \mu \in M\colon \mu \sim \mu^e$. ∘

Note 3.5.4. The SPARQL Query is only a special case of the incomplete SPARQL Query, with $dom^e(Q) = \emptyset$. ∘

```
{
  ?x dc:title ?title.
  OPTIONAL {
    ?x ns:price ?price .
    FILTER (bound(?hasPrice))
  }
}
```

Figure 3.2: Sample SPARQL Query

Note 3.5.5. In the sample query in the figure 3.2 there is a SPARQL query Q with following attributes:

- $dom(Q) = \{?\texttt{x}, ?\texttt{title}, ?\texttt{price}\}$ - these variables may be bound
- $dom^o(Q) = \{?\texttt{hasPrice}\}$ - this variable is used, but it cannot be bound
- $dom^e(Q) = \{?\texttt{price}, ?\texttt{hasPrice}\}$ - these variables are used and may not be bound, that means that the evaluation may be affected by an external value (from other part of a larger query)

∘

We have introduced the complete block of the SPARQL Query, and now we need to define the building blocks of the query. We will proceed from the basic elements to the solution modifiers. The following operators represents a SPARQL Query. However, most of them are using another SPARQL Queries as operands. So we can imagine the SPARQL Query as a tree, which nodes are operators (leafs are the operators that do not have any SPARQL Query as an operand). Every subtree of such tree is also a SPARQL Query (nevertheless it may be incomplete).

Definition 3.6. The **Basic graph pattern** BGP is a mapping $BGP\colon DS \to \mathcal{P}(M_{BGP})$ where DS is the source dataset, and $\mathcal{P}(M_{BGP})$ is the set of all possible solution sequences with variables used in the pattern. The set of used variables is the domain $dom(BGP)$. The basic graph pattern does not have any variables out of the scope, so the $dom^o(BGP) = \emptyset$. The **empty graph pattern** is a mapping $EGP\colon G \to \{M_E\}$ where M_E is the solution sequence containing one solution mapping $\mu_E\colon \emptyset \to \emptyset$. □

Note 3.6.1. The basic graph pattern is a SPARQL Query representing a triples pattern. For example, the triples pattern `?x foaf:name ?name` is represented by the BGP with domain $dom(BGP) = \{?x, ?name\}$. ∘

To cover group graph patterns, we need to introduce a way, how to combine multiple basic graph patterns into one group graph pattern. However, we will not introduce it as a combination of basic graph patterns; we introduce combinations of two queries.

Definition 3.7. The **join** operator $\bowtie$ is a binary operator to join two incomplete queries into one. For $Q' = Q_1 \bowtie Q_2$, where Q', Q_1 and Q_2 are incomplete SPARQL queries, we denote $Q'(DS, \mu'^e) \rightarrow M'$. DS is the source dataset and μ'^e is the concrete solution mapping of external variables. The M' is the set of $\mu_1 \bowtie \mu_2$ for every μ_1 and μ_2 meeting following conditions:

- $\mu_1 \sim \mu_2$
- $\mu_1 \sim \mu'^e$
- $\mu_2 \sim \mu'^e$
- $\mu_1^e = \rho(\mu_2 \bowtie \mu'^e, dom^e(Q_1))$
- $\mu_2^e = \rho(\mu_1 \bowtie \mu'^e, dom^e(Q_2))$
- $\mu_1 \in Q_1(DS, \mu_1^e)$
- $\mu_2 \in Q_2(DS, \mu_2^e)$

□

Note 3.7.1. The resulting solution mappings in M' has no additional variables, on the contrary, the result can have the smaller set of external variables. So $dom(Q') = dom(Q_1) \cup dom(Q_2)$ and $dom^e(Q') = (dom^e(Q_1) \cup dom^e(Q_2)) \setminus dom(Q')$. ∘

This join operator works exactly in the way how the group graph pattern combines the simple patterns - it joins every pair of compatible (the shared variables are bound to same values) solution mappings from two subqueries. For example, the group graph pattern in the figure 3.3 can be translated into the query $BGP_{\{?x\ foaf:name\ ?name\}} \bowtie BGP_{\{?x\ foaf:mbox\ ?mbox\}}$. We can also add constraints to

```
{
  ?x foaf:name ?name .
  ?x foaf:mbox ?mbox .
}
```

Figure 3.3: Simple group graph pattern

```
{
  ?x foaf:name ?name .
  ?x foaf:mbox ?mbox .
  FILTER regex(?name, "Smith")
}
```

Figure 3.4: Group graph pattern with filter

the group graph pattern (as in the query in the figure 3.4) and for this reason, we will introduce the selection operator. However, we will first define three expressions that will be used to define the selection operator.

Definition 3.8. The **filter expression** is a mapping $f\colon \mathcal{P}(V_x) \to \{\texttt{true}, \texttt{false}\}$ where V_x is the set of variables ($V_x \subseteq V$) and $\mathcal{P}(V_x)$ is the set of all solution mappings μ_i which have domain $dom(\mu_i) = V_x$. The set V_x is called the out of the scope domain of f, $dom^o(f)$. □

Definition 3.9. The **exists expression** is the mapping $f^E\colon \mathcal{P}(V_x) \times \mathcal{P}(Q) \to \{\texttt{true}, \texttt{false}\}$ where V_x is the set of variables ($V_x \subseteq V$), and $\mathcal{P}(V_x)$ is the set of all solution mappings μ_i which have domain $dom(\mu_i) = V_x$ and $\mathcal{P}(Q)$ is the set of all queries Q_i for which the condition $dom^o(Q_i) \subseteq V_x$ and $V_x \subseteq (dom(Q_i) \cup dom^o(Q_i))$ holds true.

$f^E(\mu, Q_i)$ is `true` exactly when for $\mu' = \rho(\mu, dom^o(Q_i))$ and $Q_i(DS, \mu') = M$ is true that $\exists \mu_i \in M\colon \mu(dom(\mu) \cap dom(Q_i)) = \mu_i(dom(\mu) \cap dom(Q_i))$.

The set V_x is called the out of the scope domain of f^E, $dom^o(f^E)$. □

Definition 3.10. The **not exists expression** is the mapping $f^{NE}\colon \mathcal{P}(V_x) \times \mathcal{P}(Q) \to \{\texttt{true}, \texttt{false}\}$ (the same as the exists expression). The $f^{NE}(\mu, Q_i)$ for $\mu \in \mathcal{P}(V_x)$ and $Q_i \in \mathcal{P}(Q)$ is true exactly when $f^E(\mu, Q)$ is `false`, $dom^o(f^{NE}) = dom^o(f^E)$. □

Note 3.10.1. The set V_x in (not) exists expression stands for variables that are used inside the filter expression and are also present anywhere outside the filter. For example the exists filter from query 3.5 has $V_x = \{\texttt{?x}, \texttt{?n}\}$. ∘

```
{
  ?x :p ?n
  FILTER NOT EXISTS {
    ?x :q ?m .
    FILTER(?n = ?m)
  }
}
```

Figure 3.5: Inner filter in exists filter

Note 3.10.2. The filter expressions contain the filtering function that will decide whether the passed solution mapping is mapped to true or to false. On the contrary, the (not) exists expressions always have the same filtering function, that is using the extra parameter (a SPARQL query). They differ only in their out of the scope domains. ◦

Definition 3.11. The **selection** operator σ is an operator to filter results of an query. We define it as the tuple of the sets containing filters and (not) exists expressions (in tuples with their corresponding inner queries), $\sigma = (\Sigma_f, \Sigma_{f^E}, \Sigma_{f^{NE}})$:

- $\Sigma_f = \{f_1, f_2, ..., f_k\}$ where f_i is a filter expression $f_i \colon \mathcal{P}(V_i^f) \rightarrow \{\texttt{true}, \texttt{false}\}$.
- $\Sigma_{f^E} = \{(f_1^E, Q_1^E), (f_2^E, Q_2^E), ..., (f_j^E, Q_j^E)\}$ where (f_i^E, Q_i^E) is the tuple of exists expression and its inner query, $f_i^E \colon \mathcal{P}(V_i^E) \times \mathcal{P}(Q) \rightarrow \{\texttt{true}, \texttt{false}\}$ and $Q_i^E \in \mathcal{P}(Q)$.
- $\Sigma_{f^{NE}} = \{(f_1^{NE}, Q_1^{NE}), (f_2^{NE}, Q_2^{NE}), ..., (f_k^{NE}, Q_k^{NE})\}$ where (f_i^{NE}, Q_i^{NE}) is the tuple of not exists expression and its inner query, $f_i^{NE} \colon \mathcal{P}(V_i^{NE}) \times \mathcal{P}(Q) \rightarrow \{\texttt{true}, \texttt{false}\}$ and $Q_i^{NE} \in \mathcal{P}(Q)$.

The selection operator is an unary operator to filter results of an incomplete query. $Q' = \sigma(Q)$, where Q' and Q are incomplete SPARQL queries, means that:

- Operator σ does not bind any additional variables, so $dom(Q') = dom(Q)$.
- Expressions in σ may use variables that can be unbound in Q, so the operator σ can add additional external variables, $dom^e(Q') \subseteq dom^e(Q)$. It can be calculated using following formula:

$$dom^e(Q') = \left(dom^e(Q) \cup \left(\bigcup_{f_i \in \Sigma_f} dom^o(f_i) \right) \cup \left(\bigcup_{(f_1^E, Q_1^E) \in \Sigma_{fE}} dom^o(f_1^E) \right) \right.$$
$$\left. \cup \left(\bigcup_{(f_1^{NE}, Q_1^{NE}) \in \Sigma_{fNE}} dom^o(f_1^{NE}) \right) \right) \setminus dom(Q)$$

- We denote $Q'(DS, \mu'^e) \rightarrow M'$. DS is the source dataset and μ'^e is the concrete solution mapping of external variables. The M' is the set of all solution mappings μ meeting following conditions:
  - $\mu^e = \rho(\mu'^e, dom^e(Q))$
  - $\mu \in Q(DS, \mu^e)$
  - $\mu \sim \mu'^e$
  - $\mu' = \mu \bowtie \mu'^e$
  - $\forall f_i \in \Sigma_f : \mu'_i = \rho(\mu', dom^o(f_i)), f_i(\mu'_i) = \texttt{true}$
  - $\forall (f_i^E, Q_i^E) \in \Sigma_{f^E} : \mu'_i = \rho(\mu', dom^o(f_i^E)), f_i^E(\mu'_i, Q_i^E) = \texttt{true}$
  - $\forall (f_i^{NE}, Q_i^{NE}) \in \Sigma_{f^{NE}} : \mu'_i = \rho(\mu', dom^o(f_i^{NE})), f_i^{NE}(\mu'_i, Q_i^{NE}) = \texttt{true}$

□

Note 3.11.1. The proposed definition of the selection operator seems complicated, but it is straightforward. The selection operator contains the set of expression and their inner queries for (not) exists expressions. The evaluation of the operator is a filter of values - all expressions must be true. ○

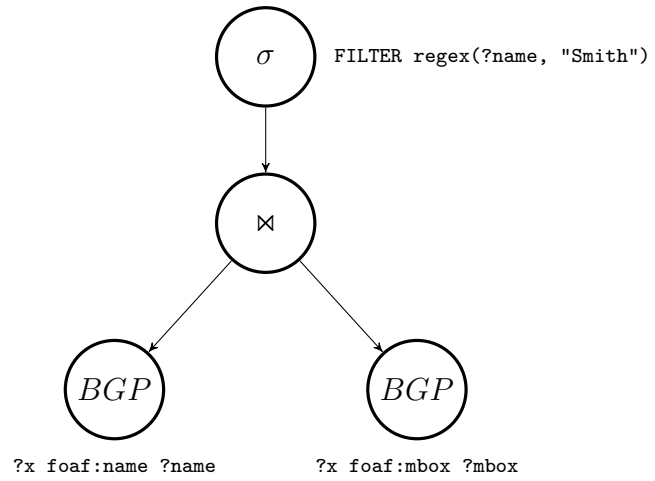


Figure 3.6: Algebra for the query in the figure 3.4

In the group graph pattern, there can also be optional values. The optional values are used only when applicable. Otherwise, it only extends the domain,

and the variables remain unbound. To handle the optional values we propose the left join operator, in a similar way as we introduced the join operator.

Definition 3.12. The **left outer join** operator $⟕_\sigma$ is a binary operator to optionally join two incomplete queries into one. For $Q' = Q_1 ⟕_\sigma Q_2$, where Q', Q_1 and Q_2 are incomplete SPARQL queries and σ is the selection operator, we denote $Q'(DS, \mu'^e) \rightarrow M'$. DS is the source dataset and μ'^e is the concrete solution mapping of external variables. The $M' = M_{join} \cup M_{notjoin}$ where $M_{join} = \sigma(Q_1 \bowtie Q_2)(DS, \rho(\mu'^e, dom^e(\sigma(Q_1 \bowtie Q_2))))$ and $M_{notjoin}$ is the set of all μ_1 meeting the following conditions:

- $\mu_1^e = \rho(\mu'^e, dom^e(Q_1))$
- $\mu_1 \in Q_1(DS, \mu_1^e)$
- $\mu_2^e = \rho(\mu_1 \bowtie \mu'^e, dom^e(Q_2))$
- $\forall \mu_2 \in Q_2(DS, \mu_2^e) \colon \mu_1 \bowtie \mu_2 \notin M_{join}$

□

Note 3.12.1. The left outer join is very similar to a standard join of two queries with additional filter (with the scope over both source queries). However, when for some solution mapping from the left operand there is no suitable solution mapping from the right operand, then also the solution mapping from the left operand is in the result, without any modifications. In other words, all solution mappings from the left operand will be present in the result, and they will be optionally extended by bindings from the right operand. ∘

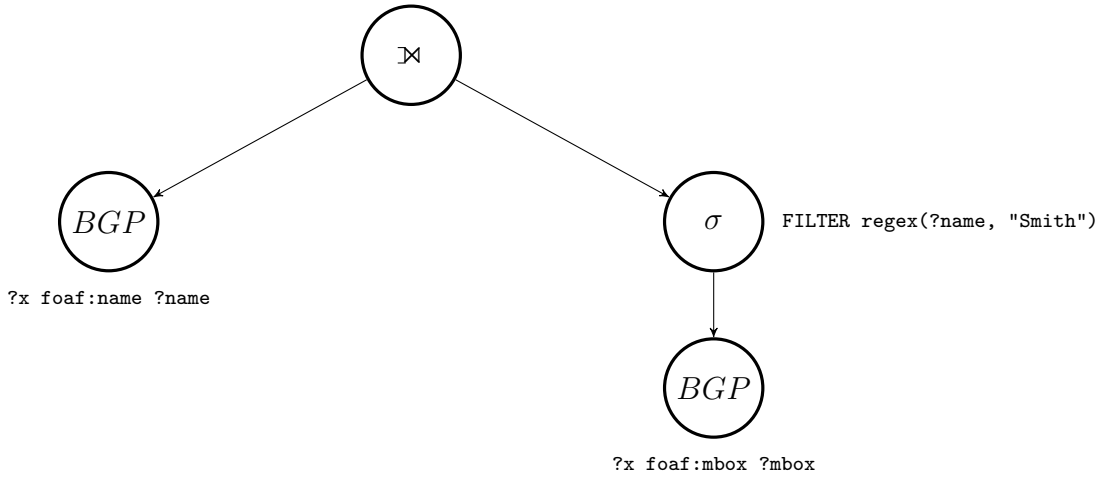
Note 3.12.2. The resulting solution mappings in M' have no additional variables in the domain. So $dom(Q') = dom(Q_1) \cup dom(Q_2)$. However, the selection operator σ can add another external variables and on top of that, all variables from Q_2 may be unbound. So the set of external variables can be calculated using the following formula:

$$dom^e(Q') = dom^e\Big(\sigma(Q_1 \bowtie Q_2)\Big) \cup \Big(dom(Q_2) \setminus \Big(dom(Q_1) \setminus dom^e(Q_1)\Big)\Big) \qquad \circ$$

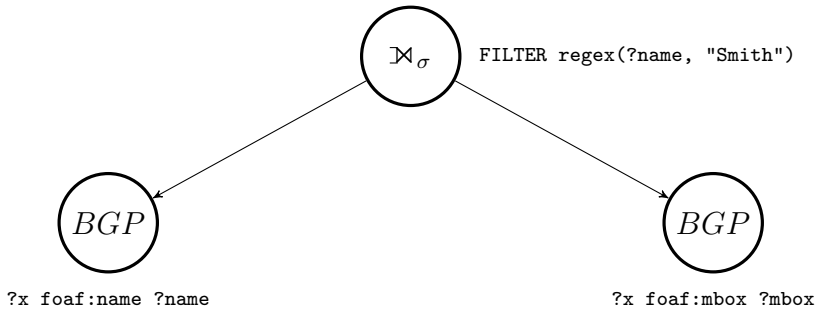
Note 3.12.3. In the figure 3.7 is the sample for the left outer join operator. It also shows the possible usage of the inner selection function, using two equivalent algebra representations (the equivalence is shown later, in the theorem 3.25). ∘

```
{
  ?x foaf:name ?name .
  OPTIONAL {
    ?x foaf:mbox ?mbox .
    FILTER regex(?name, "Smith")
  }
}
```

(a) The query



(b) The algebra



(c) The algebra transformed using the theorem 3.25

Figure 3.7: Sample algebra containing the left outer join

In SPARQL, there is possible to assign a value to a variable using BIND clause. It defines the calculation of value and the variable to be assigned. This possibility is covered in the following definition.

Definition 3.13. The **extend** operator $\varepsilon_{(?\mathtt{x};f)}$ is an unary operator to assign a value calculated in expression f to the variable ?x. The expression f is the mapping $f\colon \mathcal{P}(V_x) \to RDF\text{-}T$ where $\mathcal{P}(V_x)$ is a set of all solution mappings μ_i which have domain $dom(\mu_i) = V_x$, the set V_x is the out of the scope domain of f denoted $dom^o(f)$.

For $Q' = \varepsilon_{(?\mathtt{x};f)}(Q)$, where Q and Q' are incomplete queries, ?x is the variable

to be assigned and f is the assigning function, we denote $Q'(DS, \mu'^e) \rightarrow M'$. DS is the source dataset and μ'^e is the concrete solution mapping of external variables. It must be true that the domain of Q does not contain the variable ?x (?x $\notin dom(Q)$). The M' is the set of all μ' meeting the following conditions:

- $val = \mu'(?\mathtt{x})$
- $\mu^e = \rho(\mu'^e \bowtie \{(?\mathtt{x}, val)\}, dom^e(Q))$
- $\exists \mu \in Q(DS, \mu^e)$ meeting the following conditions:
  - $\mu' = \mu \bowtie \{(?\mathtt{x}, val)\}$
  - $\mu_c = \rho(\mu \bowtie \mu'^e, dom^o(f))$
  - $val = f(\mu_c)$

The *val* is the assigned value of the variable. Actually, it is possible that the assigning function will not return any value. In that case we understand *val* as the unbound indicator. Then $\mu^e = \rho(\mu'^e, dom^e(Q))$, $\mu' = \mu$ and $f(\mu_c)$ does not return any value. □

The BIND clause is not the only way how to assign a value to a variable. It is also possible to use VALUES clause to specify multiple values for one or more variables. Thus, we propose a definition for a query that has constant result (the specified values).

Definition 3.14. The **values** query is a query $V_M \colon DS \rightarrow M$ where DS is the source dataset, and M is the selected solution sequence that is always returned as the result. The domain of V_M is the set of used variables in the solution sequence M, so $dom(V_M) = dom(M)$. It does not have any variables out of the scope, so $dom^o(V_M) = \emptyset$. However, it is possible to let a variable unbound. The set of external variables is the set of all possibly unbound variables, so it can be calculated using the formula $dom^e(V_M) = \bigcup_{\mu \in M} \Big(dom(M) \setminus dom(\mu)\Big)$. □

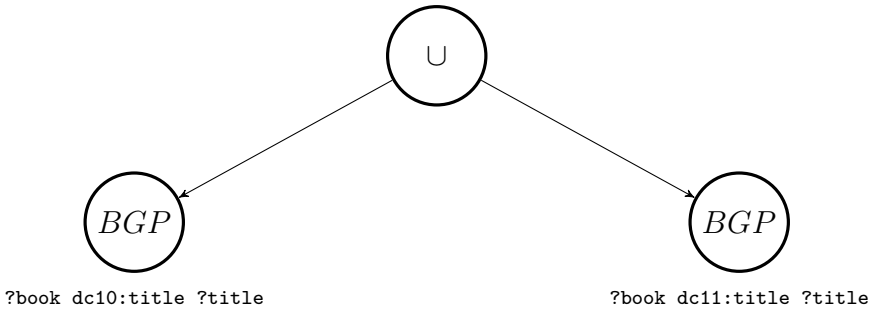
The SPARQL introduces two operators, that do the set operations on query results. The union operator and the minus operator. These operators do exactly what is expected according to their names.

Definition 3.15. The **union** operator $\cup$ is a binary operator to union two queries. For $Q' = Q_1 \cup Q_2$, where Q', Q_1 and Q_2 are incomplete SPARQL

Queries, we denote $Q'(DS, \mu'^e) = M'$. DS is the source dataset and μ'^e is the concrete solution mapping of external variables. $M' = M_1 \cup M_2$. $M_1 = Q_1(DS, \rho(\mu'^e, dom^e(Q_1)))$ and $M_2 = Q_2(DS, \rho(\mu'^e, dom^e(Q_2)))$. □

```
{
  { ?book dc10:title  ?title }
  UNION
  { ?book dc11:title  ?title }
}
```

(a) The query



(b) The algebra

Figure 3.8: Sample algebra containing the union

Note 3.15.1. The union operator makes the simple union of the solution sequences, so it affects the domain in the same way. $dom(Q') = dom(Q_1) \cup dom(Q_2)$ and $dom^e(Q') = dom^e(Q_1) \cup dom^e(Q_2)$. ∘

Definition 3.16. The **minus** operator $\setminus$ is a binary operator to filter the result from left operand using the result from right operand. For $Q' = Q_1 \setminus Q_2$, where Q', Q_1 and Q_2 are incomplete SPARQL Queries, we denote $Q'(DS, \mu'^e) = M'$. DS is the source dataset and μ'^e is the concrete solution mapping of external variables. The M' is the set of all solution mappings μ_1 meeting the following conditions:

- $\mu_1^e = \rho(\mu'^e, dom^e(Q_1))$
- $\mu_2^e = \rho(\mu'^e, dom^e(Q_2))$
- $\mu_1 \in Q_1(DS, \mu_1^e)$
- $\forall \mu_2 \in Q_2(DS, \mu_2^e) \colon \mu_1 \not\sim \mu_2 \vee dom(\mu_1) \cap dom(\mu_2) = \emptyset$

□

Note 3.16.1. The minus operator removes all solution mappings from the left operand that are compatible with any solution mapping from the right operand and they share at least one variable. So when the $dom(Q_1) \cap dom(Q_2) = \emptyset$ then $Q_1 = Q_1 \setminus Q_2$. ∘

Note 3.16.2. The minus operators does not add any variables to the resulting solution sequence. So $dom(Q') = dom(Q_1)$. But the query Q_2 may add some external variables, so $dom^e(Q') = dom^e(Q_1) \cup dom^e(Q_2)$. ∘

The group graph pattern also provides the capability to access the named graphs from RDF dataset. The GRAPH clause changes the current active graph, that will be used for matching basic graph patterns in the subquery of the GRAPH clause.

Definition 3.17. The **graph** operator is an unary operator Γ_u. $\Gamma_u(Q)$ for $Q\colon DS \to \mathcal{P}(M_Q)$ where DS is the dataset $\{G, (u_1, G_1), (u_2, G_2), ..., (u_n, G_n)\}$ and $u \in \{u_1, u_2, ..., u_n\}$ returns an identical query, the only difference is, that all basic graph patterns in the query Q will be evaluated against graph G_u. When no graph operator is used, the basic graph patterns are evaluated against the default graph G. □

Note 3.17.1. The graph operator does not do anything with variables, so the domain and the set of external variables remain unchanged. ∘

The proposed definition is for the GRAPH clause, which does not have a variable. The graph is restricted by an IRI name. For the GRAPH clause with variable instead of IRI restriction, we define the vargraph operator.

Definition 3.18. The **vargraph** operator is an unary operator $\Gamma_{\{?\mathsf{g}\}}$ ($?\mathsf{g}$ is a variable used in graph clause) and it is defined using the graph operator (from definition 3.17). For $DS = \{G, (u_1, G_1), (u_2, G_2), ..., (u_n, G_n)\}$ the vargraph operator is defined using the following formula: $\Gamma_{\{?\mathsf{g}\}}(Q) = \bigcup_{u \in \{u_1, ..., u_n\}} \left(\Gamma_u(Q) \bowtie V_{\{(?\mathsf{g}, u)\}}\right)$. The $V_{\{(?\mathsf{g},u)\}}$ is the values query that maps dataset into the solution sequence containing only one solution mapping $?\mathsf{g} \to u$. □

3.2 Allowed query parts operations

The query parts allow several operations that do not change the result of the query. These operations could be used to optimize the transformation to the

SQL query or even make it possible to generate the SQL query. We do not list every possible operation, only the ones that will be used in this work.

Theorem 3.19. *The **join** operator is associative and commutative.*

Proof. First we will prove that the join operator is commutative. The $Q_1 \bowtie Q_2 = Q_2 \bowtie Q_1$, if and only if $DS \xrightarrow{Q_1} M_1, DS \xrightarrow{Q_2} M_2$ and $\forall \mu_1 \in M_1, \forall \mu_2 \in M_2, \mu_1 \sim \mu_2 \colon \mu_1 \bowtie \mu_2 \equiv \mu_2 \bowtie \mu_1$ and $\forall \mu_1 \in M_1, \forall \mu_2 \in M_2, \mu_1 \not\sim \mu_2 \colon \mu_2 \not\sim \mu_1$. If $\mu_1 \sim \mu_2$ then $\mu_2 \sim \mu_1$ according to definition. So $\mu_1 \bowtie \mu_2$ is defined if and only if $\mu_2 \bowtie \mu_1$ is defined. The domain is equal for both variants because the operator set union is also commutative. Moreover, according to the definition, the resulting solution mapping maps the shared variables to the same values and the rest is mapped independently on the fact, whether it is from left or right operand.

The proof of the associative property is in the same way. We need to show, that $(\mu_1 \bowtie \mu_2) \bowtie \mu_3 \equiv \mu_1 \bowtie (\mu_2 \bowtie \mu_3)$. When $\mu_1 \sim \mu_2$ and $(\mu_1 \bowtie \mu_2) \sim \mu_3$ then $\forall v \in dom(\mu_2) \colon v \notin dom(\mu_3) \vee \mu_2(v) = \mu_3(v)$ because $\mu_2(v) = (\mu_1 \bowtie \mu_2)(v)$ according to definition. The same way it is true that $\forall v \in dom(\mu_1) \colon v \notin dom(\mu_3) \vee \mu_1(v) = \mu_3(v)$. So $\mu_1 \sim \mu_3, \mu_2 \sim \mu_3$. Because of that, it is true that $\forall v \in dom(\mu_2 \bowtie \mu_3) \colon v \notin dom(\mu_1) \vee (\mu_2 \bowtie \mu_3)(v) = \mu_1(v)$. So we have shown, that if and only if $\mu_1 \sim \mu_2$ and $(\mu_1 \bowtie \mu_2) \sim \mu_3$ then $\mu_2 \sim \mu_3$ and $\mu_1 \sim (\mu_2 \sim \mu_3)$. And if it is true then $(\mu_1 \bowtie \mu_2) \bowtie \mu_3$ and $\mu_1 \bowtie (\mu_2 \bowtie \mu_3)$ are defined and equal. □

Note 3.19.1. The proof is simplified to work only with complete queries. The proof for incomplete queries could be done the same way, the only difference is, that it will also work with the external variables. ∘

Theorem 3.20. *The **union** operator is associative and commutative.*

Proof. The union operator is defined as a set operation on the resulting solution sequences. The set union operation is associative and commutative, so the union operator is also associative and commutative. □

Theorem 3.21. *The **join** operator is distributive over the **union** operator.*

Proof. The join operator is commutative, so we need only to show, that it is left-distributive. We need to prove that $Q_1 \bowtie (Q_2 \cup Q_3) = (Q_1 \bowtie Q_2) \cup (Q_1 \bowtie Q_3)$. Denote $Q = Q_1 \bowtie (Q_2 \cup Q_3)$ and $DS \xrightarrow{Q} M, DS \xrightarrow{Q_1} M_1, DS \xrightarrow{Q_2}, DS \xrightarrow{Q_3}$ $, DS \xrightarrow{Q_2 \cup Q_3} M', Q^2 = Q_1 \bowtie Q_2, DS \xrightarrow{Q^2} M^2, Q^3 = Q_1 \bowtie Q_3, DS \xrightarrow{Q^3} M^3$. Then for all $\mu \in M$ is true that it as product of $\mu_1 \bowtie \mu'$ where $\mu' \in M'$ and $\mu_1 \in M_1$.

But according to the definition of union $M' = M_2 \cup M_3$, μ' is either in M_2 or in M_3 and therefore μ is either in M^2 or in M^3. So $M \subseteq M^2 \cup M^3$.

$\forall \mu^2 \in M^2$ is true that μ^2 is a product of $\mu_1 \bowtie \mu_2$ where $\mu_1 \in M_1$ and $\mu_2 \in M_2$. Because $M_2 \subseteq M'$, $\mu_2 \in M'$ and therefore $\mu^2 \in M$. We have proven that $M^2 \subseteq M$, analogically we can show that $M^3 \subseteq M$ and for that reason is true that $M^2 \cup M^3 \subseteq M$. □

Note 3.21.1. The proof is simplified to work only with complete queries. The proof for incomplete queries could be done the same way, the only difference is, that it will also work with the external variables. ∘

Theorem 3.22. *The* ***selection*** *operator can ascend over the join operator, so* $(\sigma(Q_1) \bowtie Q_2) = \sigma(Q_1 \bowtie Q_2)$.

Proof. When we filter results from one operand, they cannot be present in the resulting join. So $\sigma(Q_1 \bowtie Q_2)$ cannot contain more solution mappings than $(\sigma(Q_1) \bowtie Q_2)$. We need to show, that it also does not exclude any extra solution mapping. That could be only if there can be any solution mapping in $\sigma(Q_1) \bowtie Q_2$ that is not present in $\sigma(Q_1) \bowtie \sigma(Q_2)$. For simplicity we assume, that the selection operator σ uses only one variable ?x. If ?x is bound in Q_1, than all incorrect values of ?x are filtered in $\sigma(Q_1)$ and the join operator cannot add another one. If v is not bound in Q_1 but it is bound in Q_2 than the selection σ in $\sigma(Q_1) \bowtie Q_2$ is evaluated according to the value from Q_2 (thanks to the definition of the join operator). If ?x is not bound in Q_1 and Q_2 then ?x is not bound in $Q_1 \bowtie Q_2$, so it stays out of the scope and the rows are filtered equally whichever way it was used, $(\sigma(Q_1) \bowtie Q_2)$, $(Q_1 \bowtie \sigma(Q_2))$ or $\sigma(Q_1 \bowtie Q_2)$. □

Theorem 3.23. *The* ***selection*** *operator can ascend over left outer join operator, but only from left operand, so* $(\sigma(Q_1) \bowtie_{\sigma'} Q_2) = \sigma(Q_1 \bowtie_{\sigma'} Q_2)$.

Proof. The proof is the same as for the join operator, the only difference is, that it cannot ascend from the right operand. That is because when we filter all values in the right operand of left outer join, than the result is not empty, it only does not process the join. □

Theorem 3.24. *The* ***selection*** *operator can ascend over the minus operator, but only from the left operand, so* $(\sigma(Q_1) \setminus Q_2) = \sigma(Q_1 \setminus Q_2)$.

Proof. The minus operator is used as a filter to the left operand. The result of $Q_1 \setminus Q_2$ cannot contain any solution mapping that is not present in the result

of Q_1 and every solution mapping in the result of $Q_1 \setminus Q_2$ is also in the result of Q_1. Every solution mapping from the result of $(\sigma(Q_1) \setminus Q_2)$ must satisfy the selection operator σ and it must not be contained by Q_2. It does not matter in which order we apply these conditions. □

Theorem 3.25. *The **selection** operator can ascend into the left outer join operator (into its inner selection operator), so* $Q_1 ⟕_{\sigma_1} \sigma_2(Q_2) = Q_1 ⟕_{\sigma'} Q_2$ *where* σ' *is a selection operator containing all filter expressions (and their corresponding queries) from both selection operators* σ_1 *and* σ_2*.*

Proof. This is true according to the definition of the left outer join operator. It does not matter whether the value is not contained in the result of the right operand, or it does not satisfy the inner selection. □

Note 3.25.1. This is the reason, why there is such thing as the inner selection of the left outer join operator. We want to have the option to ascend the selection operator as much as possible to have all variables in the scope. ∘

Theorem 3.26. *The **extend** operator can ascend over the join operator, but only if the other operand has not the variable in the domain. So* $(\varepsilon_{(?\mathtt{x};f)}(Q_1) \bowtie Q_2) = \varepsilon_{(?\mathtt{x};f)}(Q_1 \bowtie Q_2)$ *when* $?\mathtt{x} \notin dom(Q_2)$*.*

Proof. The condition that the operand does not have the variable in the scope assure us that also the join $Q_1 \bowtie Q_2$ does not have the variable in the scope. The rest of the proof is simple. It does not matter in which step we introduce the variable to the solution mapping, in both cases it is added with the same value. Also, it cannot influence the join because the variable is out of the scope both Q_1 and Q_2. □

Theorem 3.27. *The **graph** operator* Γ_u *can descend to all basic graph patterns in its subquery. So the query* $\Gamma_u(Q)$ *can be changed to* Q' *where are all basic graph patterns* BGP *are replaced with* $\Gamma_u(BGP)$*.*

Proof. The proof is only rewriting definition of graph operator. The graph operator only changes the graph that is used to evaluate the basic graph pattern. So when we apply it to every BGP in the subtree of Γ_u then we can simply remove it from the root. □

3.3 Query result modifications

We have already proposed the parts that are used to build up the query. The result modifications can be understood as an operator, taking a query as an operand. However, when the solution modifier is applied to a query, the evaluation does not depend on any external value, even in the case, that the inner query was incomplete.

3.3.1 Aggregation

The SPARQL language offers a way to aggregate results in a very similar way as it is done in the SQL language. To divide the solution into one or more groups, we can use the GROUP BY clause. In the language definition [2] there is mentioned that every solution set is understood as a grouped solution - when not changed, it consists of a single group, containing all solutions. So if we do not use the GROUP BY clause, but we use aggregates as the result of the query, then the solution sequence is taken to be a single group containing all solutions.

Definition 3.28. The **aggregation** operator A_X groups the solution sequence by the variables from set $X \subseteq V$. $Q' = A_X(Q)$ means that:

- the set of all grouped variables $X \subseteq dom(Q)$
- the set of all not grouped variables $X' = dom(Q) \setminus X$
- the aggregates variables $F = \{(v, f) | v \in X', f \in \mathcal{P}(F)\}$
- the group aggregates F_G is a mapping $F_G = \{(v, f) \to f(G, v) | (v, f) \in F\}$ where $f(G, v)$ is an aggregate f of a variable v in a group G
- the domain of $Q' = X \cup F$
- for $DS \xrightarrow{Q} M$, $DS \xrightarrow{Q'} M'$
  - the set of all keys $K' = \{\mu' | \mu \in M', \mu' = \rho(\mu, X)\}$
  - the set of distinct keys $K \subseteq K'$: $\forall k' \in K' \exists k \in K : k' \equiv k$ and $\forall k_1, k_2 \in K : k_1 \not\equiv k_2$
  - the group with key k, $G_k = \{\mu | \mu \in M, \mu \sim k\}$
  - the result $M' = \{k \bowtie F_{G_k} | k \in K\}$

□

Note 3.28.1. The aggregate functions ($\mathcal{P}(F)$) defined in version 1.1 of SPARQL language [2] are COUNT, SUM, MIN, MAX, AVG, GROUP_CONCAT and SAMPLE. ○

The provided definition of aggregation operator groups the results and it offers the aggregated variables as the standard variables (with a unique name), so we do not need to define the HAVING clause, because it is the same as selection operator (except it can use only in scope variables).

3.3.2 The solution modifiers

The last applied operators are the solution modifiers. They are used after pattern matching (and aggregation if present) in the following order:

- ORDER BY
- PROJECTION
- DISTINCT
- REDUCED
- OFFSET
- LIMIT

Definition 3.29. The **cardinality** of an solution mapping μ in a solution sequence M, $card[M](\mu) = |\{\mu' | \mu' \in M : \mu' \equiv \mu\}|$. □

Note 3.29.1. The cardinality notes how many equal solution mappings are present in the solution sequence. ○

Definition 3.30. The **OrderBy** operator orders the resulting solution sequence. $Q' = OrderBy(Q, order)$ means that for $DS \xrightarrow{Q} M$ and $DS \xrightarrow{Q'} M'$ is $M' = \{\mu | \mu \in M\}$ and the M' is ordered to satisfy the ordering condition $order$. □

Note 3.30.1. The OrderBy does not change the cardinality of solution mappings, $card[OrderBy(Q, order)](\mu) = card[Q](\mu)$ and also it does not change the domain $dom(OrderBy(Q, order)) = dom(Q)$. ○

Definition 3.31. The **Projection** operator selects and possibly renames the variables in the resulting solution sequence. It is marked $Project(Q, PV)$ where $PV\colon V \to V'$ is a mapping from variables to variables. PV may not be defined for every $v \in V$. $Project(Q, PV)$ means that for $DS \xrightarrow{Q} M$ and $DS \xrightarrow{Q'} M'$ is $M' = \{\mu' | \mu \in M, \mu' = Proj(\mu, PV)\}$. $Proj(\mu, PV) = \{v' \to x | v \xrightarrow{\mu} x, v \xrightarrow{PV} v'\}$. The Project operator must not change the order of the solution sequence. □

Note 3.31.1. The Project does not change the cardinality of solution mappings, $card[Project(Q, PV)](Proj(\mu, PV)) = card[Q](\mu)$, but it does change the domain $dom(Project(Q, PV)) = \{v' | v \in dom(Q), v \xrightarrow{PV} v'\}$. ∘

Definition 3.32. The **Distinct** operator filters the solution sequence that there will be no equal solution mappings. $Q' = Distinct(Q)$ means that for $DS \xrightarrow{Q} M$ and $DS \xrightarrow{Q'} M'$ is $M' \subseteq M$ for which is true that $\forall \mu \in M\ \exists \mu' \in M'\colon \mu \equiv \mu'$ and $\forall \mu_1, \mu_2 \in M'\colon \mu_1 \not\equiv \mu_2$. The Distinct operator must not change the order of the solution sequence. □

Note 3.32.1. The Distinct does not change the domain $dom(Distinct(Q)) = dom(Q)$ but it changes the cardinality of solution mappings $card[Distinct(Q)](\mu) = 1$. ∘

Definition 3.33. The **Reduced** operator filters the solution sequence, but not so strictly as the distinct operator. $Q' = Reduced(Q)$ means that for $DS \xrightarrow{Q} M$ and $DS \xrightarrow{Q'} M'$ is $M' \subseteq M$ for which is true that $\forall \mu \in M\ \exists \mu' \in M'\colon \mu \equiv \mu'$. The Reduced operator must not change the order of the solution sequence. □

Note 3.33.1. The Reduced operator does not change the domain $dom(Reduced(Q)) = dom(Q)$ but it may change the cardinality of solution mappings. The cardinality is not strictly set, $1 \leq card[Reduced(Q)](\mu) \leq card[Q](\mu)$. ∘

Definition 3.34. The **Slice** operator is used for both OFFSET and LIMIT clause. $Q' = Slice(Q, start, length)$ means that for $DS \xrightarrow{Q} M$ and $DS \xrightarrow{Q'} M'$ is $M' \subseteq M$, where the first $start - 1$ mappings are skipped and it contains the following mappings, but not more than $length$. The Slice operator must not change the order of the solution sequence. □

Note 3.34.1. The Slice operator does not change the domain $dom(Slice(Q)) = dom(Q)$ but it may change the cardinality. The cardinality is not strictly set, $1 \leq card[Distinct(Q)](\mu) \leq card[Q](\mu)$. ∘

The provided definitions cover all operators introduced in the SPARQL language, but we did not distinguish between the query forms. We described the SELECT form. However, it does not change anything - the forms CONSTRUCT,

ASK and DESCRIBE may be modeled using the SELECT form (it only needs to represent the result in another way - as a boolean value or an RDF graph).

4. Transforming SPARQL query to SQL query

In this chapter, we will describe the process of the transformation of a SPARQL query to an SQL query and then the conversion of the results of the SQL query back to the solution mappings corresponding to the SPARQL query. The implemented tool offers only partial support for the SPARQL query. It provides the support for the basic graph patterns with some additions. So therefore we will focus on the parts that are implemented. However, we will also discuss several problems that must be handled for the not implemented parts of the SPARQL query.

The SPARQL algebra is defined in a recursive way, as a tree where every node is a SPARQL query. Also, the tree root is possibly modified by several result modifications. In this document, we try to propose an algorithm that will respect the recursive nature and will be able to generate a corresponding SQL query for every subtree. Proposed algorithm is inspired by the translation algorithm from [7] however it had not worked with the possibilities of the R2RML language and it do not fully cover the operators in SPARQL (and the problems they can cause).

4.1 Transformation phases

The query transformation can be separated into several phases. To be able to work with the SPARQL query, we need to transform it into its algebra representation. That allows us to use operations that are introduced in the section 3.2. Because our algebraic representation is very similar to the algebra given in [2] we can use the standard approach to generate the algebra (also presented in [2]).

The next phase is the transformation of the algebraic representation into a correct form, the form from which it can be converted into an SQL query. The transformation goes through the algebraic tree from leafs to root, so we need to be able to create an SQL query representing any node from the tree. That is possible only when we meet every condition that is needed for creating the SQL query, as mentioned in section 4.4.

Now we have the correct form of the algebra, but we are not still able to create the SQL query. Till this point, there has not been used any information we have from the R2RML mapping. Therefore in the following step we transform the algebra while adding the information from the mapping. After that, we have enough information to know which table we should query and how the query will look like.

At this point, we are able to transform the algebra, but first we will optimize the SPARQL algebra (using approaches described in section 5.1) and only then we will transform it into the SQL query. The SQL query form does not hold only the query information, but it also has to contain the functions that will be used to convert the query result back to the SPARQL form (so-called value binders).

Before the execution of the SQL query form, we will run optimizations that are specific for this form (as described in section 5.2). Finally, the optimized SQL query is executed using selected RDBMS. That is not the end of the transformation, after that we will need to transform the query result to the form that is expected from the SPARQL query, according to the query type.

4.2 Value binders

For every created SQL query (and every subquery) we need to know how to reconstruct the SPARQL variables. So for every SPARQL variable we attach exactly one value binder that holds the information how to construct the SPARQL value according to the column values from the SQL result. The value binder does not only contain a function that will be used after executing the SQL query, but we also use it to construct SQL expression when it is needed. So for every SQL query (and every subquery) we hold also the set of value binders used.

The value binders can get quite complex. When processing the query they can contain other value binders and choose which to use according to some condition (that can be written in SQL using the CASE statement), or take first bound variable (in SQL written using the COALESCE statement). Even when the value binder does not contain other value binders it can still reference multiple columns (for example when it represents a variable that is constructed using template, in SQL written using the CONCAT expression), or it can also reference no column (when the variable is a constant).

In some cases, it may be needed (or maybe it could optimize the query performance) if we create the variable value using the SQL statement. In SELECT statement, we can use the value binders expression to create a new calculated column that will contain the exact variable value. So the value can be then easily used. In that case, we need to replace the old value binder with a new one that will read the exact variable value from a single column. In some cases we will need to add another extra column, that will hold the information of the variable type, for example in a case when the variable can have various datatype (or even when it can be both literal and IRI).

4.3 Adding the R2RML mapping information to the algebra

The algebraic representation of the SPARQL query does not have any connection to a relational database, so we will not be able to decide which table we should query. That is changed in this phase. The operator that means the actual query to the dataset is the basic graph pattern (definition 3.6). The BGP operator means a query to all triples in the dataset that is possibly filtered. According to the R2RML definition, that means to query all possible combinations of the graph, subject, predicate and object mappings.

Definition 4.1. The **Restricted basic graph pattern** is the BGP operator restricted by a tuple $< g; s; p; o >$ of the graph, subject, predicate and object mapping from the R2RML mapping file. $BGP_{<g;s;p;o>}: DS \rightarrow \mathcal{P}(M_{BGP})$ and for every solution mapping $\mu \in BGP_{<g;s;p;o>}(DS)$ is true that $\mu \in BGP$ and it is generated according to the R2RML mapping using the graph g, subject s, predicate p and object o mapping. □

According the definition of the R2RML mapping (as presented in [3]), it generates a query for every tuple $< g; s; p; o >$ so we can take the nonrestricted basic graph pattern as the union of the restricted ones. So therefore we will replace every basic graph pattern using the rule $BGP = \bigcup_{<g;s;p;o> \in R2RML} BGP_{<g;s;p;o>}$.

After this replacement, we know the exact SQL query for every restricted basic graph pattern. Because every tuple $< g; s; p; o >$ is from an R2RML triples map, that has the `rr:logicalTable` node. In that node is stored the needed query. Also, when the o is an object map with reference to another triples map,

we know the exact triple map, that is referenced. From that triple map, we can also get the SQL query using the `rr:logicalTable` node. So we know all the SQL queries needed for this particular $BGP_{<g;s;p;o>}$.

4.4 Creating the SQL query

In this section, we will describe the transformation of the SPARQL operators and result modifiers.The creation of the SQL query works in a recursive way, we handle every operator after we have transformed its child nodes. Result of the transformation is not only the corresponding SQL query, but also information that will be used to reconstruct the SPARQL values on the basis of the SQL query execution result. This information is called a value binder and for every variable there is not more than one value binder.

4.4.1 The basic graph pattern

Because our algebra is modified, in a way, that every basic graph pattern is restricted, we know the exact SQL query (from R2RML mapping) that will generate all triples that conform the restriction. That can be in two possible forms. A simple select clause from a table (or a statement) or select clause with inner join in the case when the object mapping is a referenced object map.

So we know the tuple $< g; s; p; o >$ that is in a triple map with defined logical table using the `rr:logicalTable` node. That node defines an SQL query (we denote it $Q1$) that can be used to retrieve the data. If the object mapping contains a reference to another triple map, we know also the other SQL query (we denote it $Q2$) and moreover, we know the join condition. In that case, the object mapping uses the columns from the $Q2$ source to generate the value (using the subject mapping of the referenced triple map). The query scheme is shown in the figure 4.1.

The basic graph pattern contains three patterns, for the subject, the predicate and the object. The pattern can contain a variable match pattern, a node match pattern and a blank node match pattern. For the subject, the predicate and the subject we do the following steps according to the type of the pattern:

```
SELECT <<used columns>>
FROM Q1
WHERE <<the conjunction of
    ↪ conditions>>
```

(a) If the object mapping does not contain a reference

```
SELECT <<used columns>>
FROM Q1
INNER JOIN Q2 ON <<the join
    ↪ condition>>
WHERE <<the conjunction of
    ↪ conditions>>
```

(b) If the object mapping contains a reference

Figure 4.1: The query scheme for the basic graph pattern

- **Variable match pattern** - We create a value binder using the corresponding R2RML mapping from the tuple $< g; s; p; o >$, and using the referenced columns from the mapping. If this SELECT statement already contains a value binder for the same variable, we add a condition to the SELECT statement, that the value must be the same as the previously created value binder. However, in that case, we do not add the value binder to the SELECT statement. If this SELECT statement does not contain such value binder, we add the value binder to the SELECT statement.
- **Node match pattern** - We create a value binder using the corresponding R2RML mapping from the tuple $< g; s; p; o >$, and using the referenced columns from the mapping. Then we add a condition to the SELECT statement, that the value must be the same as the value in the node match pattern. The value binder is here used only to create the condition.
- **Blank node match pattern** - The blank node pattern is handled like the variable pattern because in the SPARQL language the blank nodes in the where clause are very similar to standard variables. They have only a different scope.

The columns in the SELECT statement is created in a way, that we take all columns that are used in the value binders in the created SELECT statement.

4.4.2 The join operator

The join operator has two subqueries that are recursively converted to an SQL statement. We assume that they will be converted into a SELECT clause. If the statement is not a SELECT clause, it can be easily wrapped into one.

The SPARQL join operator is semantically the same as the SQL join. The most straightforward way, how to transform it, is to take the SELECT clause from the first subquery and add the SELECT clause from the second subquery as the last joined source (in the form of the subselect statement). To create the join condition, we need to take the value binders for variables that are present in both of the subqueries. Then for every value binder from this collection we have a condition that the value is equal, or the variable remains from at least one of the subqueries unbound.

```
1 SELECT <<used columns>>
2 FROM Q1
3 INNER JOIN Q2 ON <<the conjunction of the join conditions>>
```

Figure 4.2: The query scheme for the join operator

The simplest approach is shown in the figure 4.2. We take the SQL statements of the left operand (we denote it $Q1$) and the right operand (we denote it $Q2$). For every SPARQL variable, that is present in the query $Q1$ and not in the query $Q2$, or it is present in the $Q2$ and not in the query $Q1$, we add its value binder to the created SELECT statement. For the variables, which are present in both queries we need to create a new value binder. For every such variable, we take the value binder from the query $Q1$ (we denote it $VB1$) and the value binder from the query $Q2$ (we denote it $VB2$). Then the created value binder is a value binder VB that works in a way, that it returns the value using the value binder $VB1$ if it is not null. Otherwise, it returns the value using the value binder $VB2$. In the SQL language, it can be written as it is shown in the figure 4.3a. This newly created value binder is then added to the created query. Moreover, for every variable that is in both queries $Q1$ and $Q2$ we need to add a condition, that the values are the same using the scheme in the figure 4.3b.

```
1 COALESCE(<<expression for value from VB1>>
2   , <<expression for value from VB2>>)
```

(a) The value binder scheme

```
1 <<value from VB1 is NULL>>
2   OR <<value from VB2 is NULL>>
3   OR <<values are equal from both VB1 and VB2>>
```

(b) The join condition scheme

Figure 4.3: The other schemes for the join operator

In some cases, it is possible to create the join without using the subselect statement. It can be joined in a flattened form - we will join the first SELECT clause with the original source (the one from the FROM clause) from the second SELECT clause with the condition created, in the same way, as it is created using the subselect. Then we will add other sources from the second SELECT clause (the sources from joins and left outer joins). We need to be aware of two issues. First of all this approach can produce a different result, because the SELECT clause can contain modifiers for the result (like LIMIT, OFFSET, aggregation and orderings). When there is some of them, we have to use the subselect because otherwise we have no chance to apply the modifier in the same way. The other issue is that we need to ensure that every column used in conditions in the SQL statement has to be accessible in the moment of use (so its source has to be declared in the statement before the first use of the column).

4.4.3 The selection operator

The selection operator is performed over an inner query which is firstly transformed into an SQL SELECT statement. The selection operator is semantically very similar to the WHERE clause in the SQL SELECT statement. In the relational query, it is also possible to create a filter and (not) exists expressions and use their conjunction (using AND). That is exactly what will be the result of the transformation. We will transform every expression in the selection operator into an SQL form and all of them we will add as a conjunction to the WHERE statement from the transformed inner query.

The filter expression transformation is straightforward. We need only to transform the operators and function in the SPARQL expression into the SQL form. However, it can be easily done only when $dom^o(f) \cap dom^e(Q) = \emptyset$ (f is the filter expression, Q is the inner expression of the selection operator). In other words, all used variables are bound in the inner query of the selection operator. Otherwise we will not be able to easily create the SQL expression for the variables that may have unknown value in the time of the transformation. Even if we know how the value binder will look like (that it will be present in some ascendant of the selection operator), we cannot use that information without assuring that all used columns are introduced before they will be used here).

The exists expressions can be transformed into the EXISTS statement. First we will transform the inner query of the exists expression, but we need to handle

the variables from $dom^o(f^E)$ (variables that are present in the inner query of the exists expression, but also outside of it). Every variable from $dom^o(f^E)$ in the Q_{fE} should be replaced using the value binder that we take from the transformation of Q (or get using the same way as in the filter expression if the variable is outside of the scope). The not exists expressions are transformed the same way but with negation of the result (NOT EXISTS statement).

As mentioned, we need to handle the problem when the $dom^o(f) \cap dom^e(Q) \neq \emptyset$. There are two possibilities how to solve this. We can transform the algebra using the allowed operations to ensure that all variables will be bound when they are used. We try to move the filter expressions upwards through the algebraic tree (this step is part of the algebra transformation into a correct form). However, it is not always possible to remove every external variable. In this case, we need to use the other approach. The other approach is more complex than the previous one, we need to find the final value binder for the variable (he should be present in some ascendant in the algebraic tree) and this value binder should be used in the transformation. We need to ensure that the columns used by the value binder are introduced in the SQL expression before they are used (this must be also done using the allowed operations).

4.4.4 The left outer join operator

The left outer join operator can be in a state that it will not return the proper results. We will correct it during the transformation phase when we are correcting the algebra. After that, we can transform the operator using the same mechanics as in the join operator transformation. However, we will use the LEFT OUTER JOIN statement instead of the INNER JOIN statement. Moreover, we need to add to the join condition also the left outer join filter (with the same transformation process used to transform the selection operator).

The nested OPTIONAL problem

The SQL left outer join statement does not have exactly the same semantic as the OPTIONAL clause. The difference appears in a case when there are two or more OPTIONALs nested. The problem is mentioned in [6] with the query in the figure 4.4 (its algebraic representation with the only one possible assignment in BGPs is in the figure 4.5).

```
x :p1 1; :p2 2; :p3 3.

SELECT ?b ?c
WHERE
{
  ?a :p1 ?b
  OPTIONAL {
    ?a :p2 ?c
    OPTIONAL { ?a :p3 ?b }
  }
}
```

Figure 4.4: The "NESTED OPTIONALSs" problem from [6]

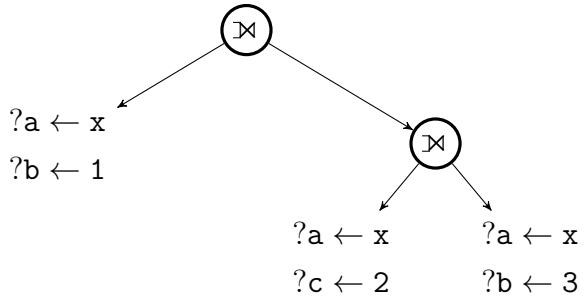


Figure 4.5: Algebraic representation of the problem 4.4

What is the expected result of the 4.4 query? There is only one possible assignment, from the triple ?a :p1 ?b we assign x to ?a and 1 to ?b. The triple inside of the first optional ?a :p2 ?c matches with ?a ← x; ?c ← 2. And that can be joined with the previously matched triple. The last triple inside the nested optional with ?a ← x; ?b ← 3 and that cannot be joined with the previous, so the nested optional will fail. So the expected result is ?b ← 1; ?c ← 2. The expected processing is in the figure 4.6.

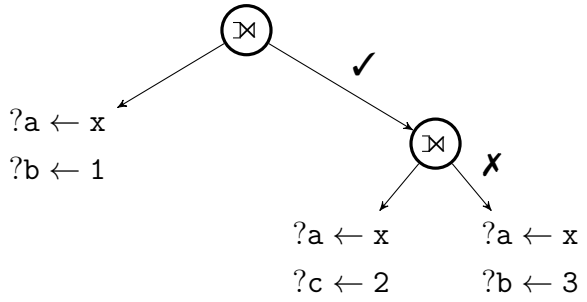


Figure 4.6: SPARQL processing of the problem 4.4

However, when we evaluate the query using SQL, we will get a different result. SQL evaluates the left outer join from bottom, so it will first join the OPTIONAL

with the nested one, that will result in the assignment ?a ← x; ?b ← 3; ?c ← 2 and when we try to join this with the assignment of the top level triple (?a ← x; ?b ← 1), it will fail because we have already assigned ?b ← 3. So we get the result ?b ← 1 and ?c is unbound. The relational processing is in the figure 4.7.

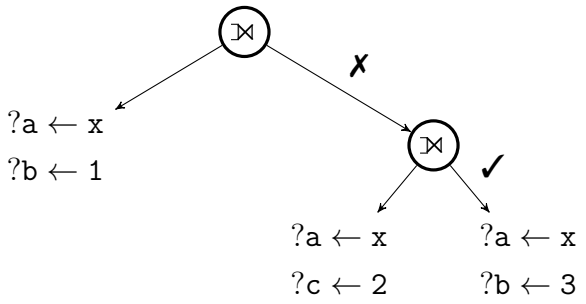
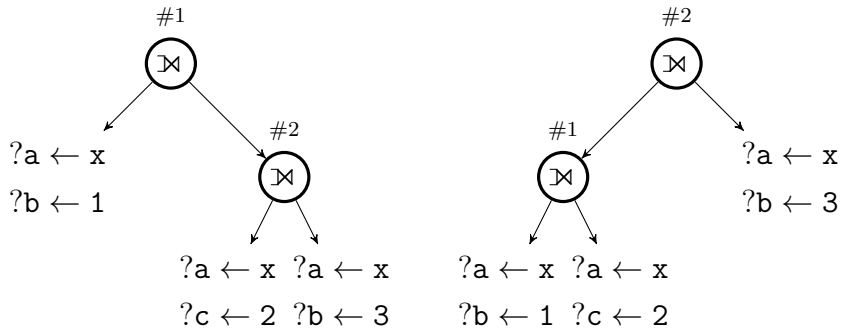


Figure 4.7: Relational processing of the problem 4.4

The problematic of the nested optionals are also described in [8], and there is also proposed a solution to this problem. We need to transform the algebra in the form that the processing from left to right will be the same as from bottom. This can be done as shown in the figure 4.8. We take the left operand of the left outer join #2 and assign him as the right operand of the left outer join #1. The left operand of the left outer join #2 will be the left outer join #1. So we changed the order of evaluation, that the from bottom means the same as the order from left to right. However, we also need to modify the condition of the left outer join #2, it can join the right operand only if the right operand of the left outer join #1 has been joined. It can be done using several approaches. In this case, we can check whether the variable ?c is bound, but it is also possible to add a calculated variable to check whether it is bound (to simplify the checking, checking an arbitrary variable could result in some complex condition, if we add a suitable variable, we will check only whether one column is NULL or not).



(a) Before (b) After

Figure 4.8: Transformation of the NESTED optional

4.4.5 The extend operator

The extend operator assigns a result of an expression into a variable. We do not need to transform it in any way, we only create a value binder (with the expression from the extend operator) for the value binder and add this created value binder into the transformed inner query. Then when we need to use the variable we know how to construct its value, but we do not need to modify the query. For the transformation, we may need to use the same mechanism as in the filter expression for the variables that are out of the scope.

4.4.6 The values operator

The values query can be transformed into an SQL SELECT statement which source will be defined using the VALUES clause. The exact form depends on the type of the RDBMs, for the T-SQL language (used in MS SQL, described in [9]) is the transformation shown in the figure 4.9. To this select statement we need to add also the value binders for the variables. They need to handle the NULL value (it results in an unbound variable) and also it has to handle the variable type. In this case, we know that the book column contains an IRI (or NULL), so the value binder take the value as an IRI. However, it can differ, it can be a literal in one row and IRI in an another. In that case, we need to add an extra column to the SQL VALUES collection that will be used to recognize the type.

```
1 PREFIX :     <http://example.org/book/>
2
3 VALUES (?book ?title)
4 {
5   (UNDEF "SPARQL Tutorial")
6   (:book2 UNDEF)
7 }
```

(a) SPARQL query

```
1 SELECT book, title
2 FROM (VALUES
3   (NULL, 'SPARQL Tutorial'),
4   ('http://example.org/book/book2', NULL)
5 ) AS vs(book, title)
```

(b) SQL query

Figure 4.9: Transformation of the values operator

4.4.7 The union operator

Although there is the union operator in the SQL language, it is not so straightforward to use it.

```
SELECT <<Q1COLS>>
FROM Q1
```

(a) The query for the left operand

```
SELECT <<Q2COLS>>
FROM Q2
```

(b) The query for the left operand

```
SELECT <<Q1COLS>>, <<Q2COLSM>>, 0 AS source_id
FROM Q1

UNION ALL

SELECT <<Q2COLS>>, <<Q1COLSM>>, 1 AS source_id
FROM Q2
```

(c) The query scheme

```
CASE
  WHEN source_id = 0 THEN <<expression for value from VB1>>
  WHEN source_id = 1 THEN <<expression for value from VB2>>
```

(d) The value binder scheme

Figure 4.10: The schemes for the union operator

The query scheme for the union operator is shown in the figure 4.10c. For simplicity, we used very simple queries for the operands, but they can be any SELECT statement. We need to add a column, so it must be a SELECT statement (otherwise, we wrap it in one, as mentioned earlier). There can be different value binders for the same variable (we have multiple possible sources for the value) so we need to decide, which value we can use. That is done using the added column to the queries (`source_id`) with specific values for every source.

Also, we need to modify the value binders. For every variable present in the query of the left (with the value binder $VB1$) or the right (with the value binder $VB2$) operand, we create a new value binder that will return the value according to the value in the column `source_id`. In the SQL language, it can be written using the scheme in the figure 4.10d. This newly created value binder is then added to the created query.

In the query schema in the figure 4.10c there are extra columns «Q1COLSM» and «Q2COLSM». That is because the SQL requires that both unioned queries

do have the same columns. So «Q1COLSM» is created in the way, that we take every column from «Q1COLS» and we add it to «Q1COLSM» in the form `NULL AS` «column name».

4.4.8 The minus operator

The minus operator in the SQL language and the SPARQL language do exactly the same thing, but it cannot be used without extra processing. Although it does the elimination of the result the same way, in the SQL language we are working with the column values and in the SPARQL language with the variable values. These variables may be represented with more columns, and the same variable value can be produced by different column values. So we can use two different approaches. We can transform the select, in a way, that the variable values will be identified by the column values. That can be done when we create the variable value exactly into a single column (and possibly one extra column holding the variable type). However, creating the variable is not so effective, the expression will probably contain CASE statement and for the variables mapped using the template also the CONCAT expression.

The other approach is not to use the MINUS statement, but to change it to selection operator, exactly the NOT EXISTS statement. This method can be used only when there is some shared variable between left and right operands (but that is not a problem because we know that if there is no variable shared then the minus operator does not remove any result). If there is shared variable, we can simply change the minus operator into the selection with not exists filter with a query taken from the right operand of the minus operator. That works if there is a shared variable because in not exists pattern we ask whether there is any solution mapping corresponding to the inner query. In that case we replace the occurrences of the shared variables by the actual values from the left operand, so with share variable we ask for any compatible mapping. Using the minus operator we ask for the compatible solution mapping. So it is the same.

4.4.9 The graph operator

Using the theorem 3.27, we can descend all the graph operators right to the BGP operator. When we have a graph operator over the BGP operator, we only need to add the condition that the used mapping of subject, predicate and object is in

the selected graph.

4.4.10 Aggregation

The aggregation system in the SPARQL language is the same as in the SQL language. To be able to use the SQL functions we need first to convert the value binders in a form where the whole variable value is in a column, in a way, in which it can be processed by the aggregation function (in case of the SUM, MIN, MAX and AVG functions). For the COUNT function, we do not need any processing (because one row in SQL result represents exactly one solution mapping in SPARQL result). The functions GROUP_CONCAT and SAMPLE do not have any analogous functions in the SQL language, but it may be possible to simulate the aggregation using functions specific to the used RDBMS. For example MSSQL offers functionality connected with XML documents, that can be used for the advanced processing of multiple rows into one.

4.4.11 The solution modifiers

The SPARQL language provides several solution modifiers. They are applied over the final SELECT clause.

The ORDER BY operator works in the same manner as in the SQL language, but we need to add columns that will be used to order in SQL. However, the columns must simulate the situation when we have a variable with more possible types. We need to follow the order introduced in the definition of SPARQL language (see [2]), that means the following order:

1. Not bound variables
2. Blank nodes
3. IRIs
4. RDF literals

The projection operator does not involve any changes in the SQL query. We only work with the value binders. Some remain untouched, some are renamed, and some are removed at all.

For the distinct operator, we need to transform the columns (and the value binders) in a way that the row will uniquely identify a solution mapping. So for every variable we will have one column with the variable value (or two columns with the variable value and type as mentioned in section 4.2). To this transformed SQL query we only add DISTINCT statement.

For the reduced operator, we can (optionally) add the DISTINCT statement too, without any transformation. It will remove all duplicities in the SQL execution result. It may not remove all duplicities in the SPARQL result set (because two different SQL results may end up in the same SPARQL solution mapping), but it does not remove any solution mapping that must not be removed.

For the slice operator, we can use the SQL statement because we have our data in a manner that one row in SQL result corresponds to exactly one solution mapping. Unfortunately, the slice operator differs in various RDBMS, so we need to produce the right statement for the used RDBMS. Moreover, also meet any restrictions of a particular RDBMS implementation, for example, MSSQL can use OFFSET only if the operator ORDER BY is used.

4.4.12 Other needed operations

There are several operations that are needed to generate the SQL query, and we have not them covered.

Property path

The triple pattern in SPARQL may not be only in the form subject, predicate and object, it can contain more complex structure for the predicate. However, these are not mentioned in our proposed algebra. That is because we can convert the property path into the simple form. In the figure 4.11, we propose the conversion of the common property path expression. There are several more, that can be converted too, but the conversion is not so plain (property sets and ZeroOrOne) and there are property paths that will be difficult to map into an SQL query (OneOrMorePath and ZeroOrMorePath). These are not yet supported by the proposed solution.

Name	Form	Transformed to
Predicate	?x *iri* ?y	-
Inverse	?x ^path ?y	?y path ?x
Sequence	?x $\texttt{path}_1$/$\texttt{path}_2$?y	(?x $\texttt{path}_1$?z) $\bowtie$ (?z $\texttt{path}_2$?y)
Alternative	?x $\texttt{path}_1$|$\texttt{path}_2$?y	(?x $\texttt{path}_1$?y) $\cup$ (?x $\texttt{path}_2$?y)

Figure 4.11: Converting the property path

The vargraph operator

In our proposed solution, we work only with a static set of graphs. In R2RML language, it is also possible to assign the graph dynamically (according to the column(s) value). Our proposed solution (to generate all possible graphs and make the union of this) can still be used, but it is not effective and there probably is some more effective approach. However, our algebra currently does not provide any better support.

4.5 Transformation of the SQL result

After we have created and executed the SQL query, we need to transform the SQL result to the form that is expected by the SPARQL query. If we have the SPARQL query in the SELECT form, we only need to use value binders for every row. Every value binder represents a variable, and every row represents a solution mapping. So that is the only processing needed.

If the SPARQL query is in the CONSTRUCT form, we need first to create solution mapping for every row from the SQL result and then apply the template from CONSTRUCT to build the triples from the solution mappings. The sample is shown in the figure 4.12. The template is applied for every returned solution mapping. The template can contain blank nodes, these blank nodes are generated unique for every single solution mapping, for different solution mappings there will be different blank nodes generated.

For the ASK form, we need only to check whether the solution has any possible solution mapping or not. Because we have our query in a form that every row from the SQL result represents a solution mapping, so we need only to check whether there is at least one row returned. To optimize this, we can improve our query using slice operator, to limit the returned result count, so we will have zero or one result.

```
CONSTRUCT { ?x  vcard:N _:v .
  _:v vcard:givenName ?gname .
  _:v vcard:familyName ?fname }
```

(a) The template

#	?x	?gname	?fname
1	_:a	Alice	Hacker
2	_:b	Bob	Hacker

(b) Variable bindings

#	subject	predicate	object
1	_:a	vcard:N	_:x1
2	_:x1	vcard:givenName	Alice
3	_:x1	vcard:familyName	Hacker
4	_:b	vcard:N	_:x2
5	_:x2	vcard:givenName	Bob
6	_:x2	vcard:familyName	Hacker

(c) Results set

Figure 4.12: Sample CONSTRUCT template processing

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
DESCRIBE ?x
WHERE { ?x foaf:name "Alice" }
```

(a) DESCRIBE form

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT { ?x ?p ?o }
WHERE
{
  ?x foaf:name "Alice";
    ?p ?o.
}
```

(b) CONSTRUCT form

Figure 4.13: Converting the DESCRIBE into the CONSTRUCT form

.

The DESCRIBE form has no exact specification what should be returned, but we can convert it into a CONSTRUCT form returning all resources that are connected with the node that should be described, example shown in the figure 4.13.

5. Optimizing query

In this chapter, we will discuss methods that can be use to optimize the query or the transformation process. We have two main types of optimization. The one focused on optimizing the SPARQL query and the other focused on optimizing SQL query. We will also discuss other options how to improve the performance.

5.1 SPARQL algebra optimization

In this section, we will propose several methods that can be used to optimize the process using some transformations on SPARQL algebra. This kind of optimization is used to simplify the algebra for the conversion into the SQL query.

We detect the cases when we are able statically to decide that the operator will not return any result. Moreover, this observation we propagate through the algebraic tree to the ascendant operator. The ascendant operator must process the information. An example is shown in the figure 5.1. Join operator, with one operand that will not return any result, also cannot return any result (step 1). Also, when one operand of the union operator will not return any result, we can replace the operator with the other operand (step 2). So the whole tree from the figure 5.1 will be replaced by the node Q_2.

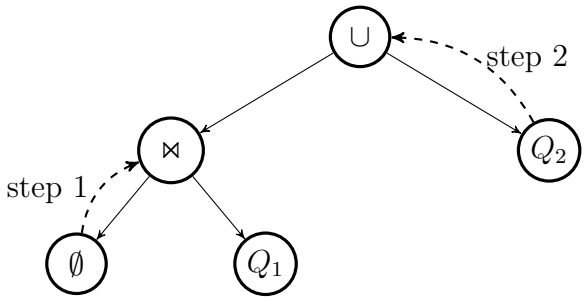


Figure 5.1: Sample processing of the "no possible result" information

5.1.1 Filtering the restricted basic graph pattern

In the section 4.3, we introduced a system how to replace the basic graph patterns by the union of the restricted graph patterns. If the basic graph pattern contains

an IRI or literal match in subject, predicate or object we can decide which of the restricted basic graph pattern can actually return some results.

For example, when there is mapping with a template we need to decide whether the template can match or not (as seen in the figure 5.2), we decide according to the possible values that can come from the column. Note that if the template results in an IRI, the column values are converted into an IRI-safe version (see [3]), so as visible in the sample #3 and #4 (from the figure 5.2) "12/45" cannot match column value, but "12-45" can. If we load the column types and for example we will get that the Code column has integer type, then we know that the sample #4 cannot match.

#	**BGP pattern**	**R2RML mapping template**	**Can match**
1	http://s.com/12345	http://s.com/{Code}	✓
2	http://s.com/12/45	http://s.com/{Code}/{Label}	✓
3	http://s.com/12/45	http://s.com/{Code}	✗
4	http://s.com/12-45	http://s.com/{Code}	✓
5	http://s.com/12-45	http://s.com/{Code}-{Label}	✓
6	http://s.com/12-45	http://s.com/{Code}/{Label}	✗

Figure 5.2: R2RML mapping matches

5.1.2 The join optimization

The other optimization method also uses the mapping information from the R2RML language. In lots of cases, we can decide whether a join can return something.

For simplicity we will show the method on two joined basic graph patterns, that share only one variable (the figure 5.2, the first line represents BGP_1, second-one BGP_2). With restricted basic graph patterns, we consider the join in the form $BGP_{1<g;s_1;p_1;o_1>} \bowtie BGP_{2<g;s_2;p_2;o_2>}$. The shared variable is ?`nuts2` as the object in BGP_1 and the subject in BGP_2. So this join can return something only if the mapping o_1 can produce the same value as the mapping s_2.

```
1 ?nuts1 ec:hasSubRegion ?nuts2.
2 ?nuts2 ec:name ?name2.
```

Figure 5.3: Sample basic graph patterns

In the common case, the method goes through all the shared variables. To improve the results it is better not to see the join as an operator with two operands, but to take nested joins as one large join and process all operands at once. That means that there may be some variable shared between more than two BGP operands, and therefore we will dismiss more operators that will not return any results.

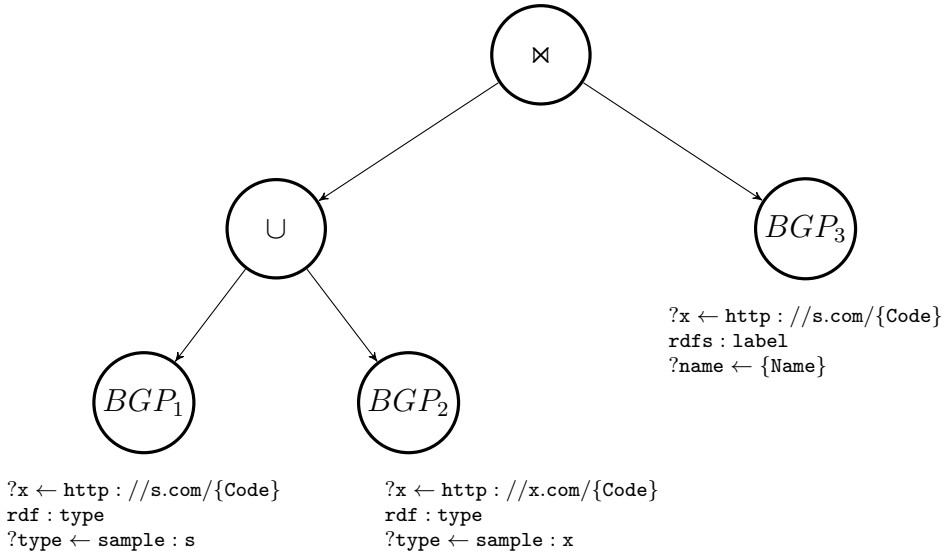
5.1.3 The union optimization

The join optimization (previous subsection, 5.1.2) works with the joins of restricted basic graph patterns. However, these restricted graph patterns are generated in a union. The join operator is usually the ascendant of these created unions, so we cannot apply the previous optimization. From the theorem 3.21, we know that $Q_1 \bowtie (Q_2 \cup Q_3) = (Q_1 \bowtie Q_2) \cup (Q_1 \bowtie Q_3)$ so we can move the union operator over the join operator.

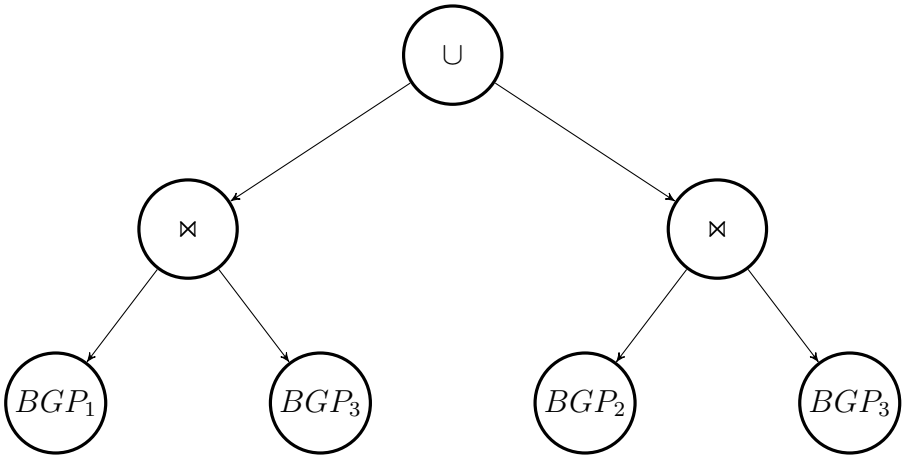
Using this approach, we can enable the use of the join optimization (mentioned in 5.1.2) so we can more precisely select the joins that can return some results. Moreover, we can achieve a more efficient form of the SQL query when we have the union of simply joined tables instead of the join of complex (unioned) sources.

A sample is shown in the figure 5.4. The triples patterns BGP_1 and BGP_2 assigns to variables `?x` and `?type`. The triples pattern BGP_3 assigns to variables `?x` and `?name`. After the application of the union optimization, the algebra is bigger, but the corresponding SQL query is slightly faster. However, the benefit is that this optimization method prepared the algebra for the join optimization. After the application of the join optimization (the templates for the variable `?x` in BGP_2 and BGP_3 cannot match), we have only a single join. The join $BGP_2 \bowtie BGP_3$ is discarded because the templates for the variable `?x` cannot match.

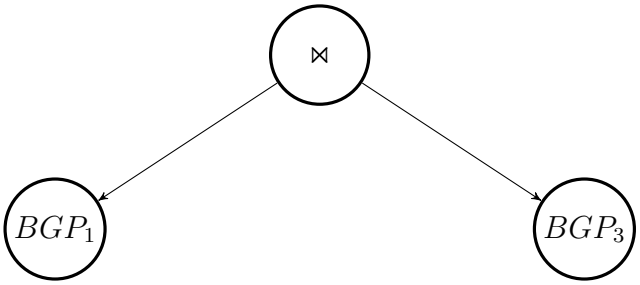
However, it also has its drawbacks. We need to reckon with the fact, that it can produce lots of unioned joins (it is actually the cartesian product of the unions). For example, when we have the join of ten triple basic graph patterns (and that is not a large count, especially when we create some from the property path patterns) and every basic graph pattern will have five valid restricted basic graph patterns (after filtering using the mechanism from the subsection 5.1.1) then we will create a union of $5^{10} = 9765625$ joins.



(a) Original algebra



(b) After the application of the union optimization



(c) After the application of the join optimization

Figure 5.4: The union optimization sample

To optimize the method, we will merge this method with the join optimization (mentioned in 5.1.2). We will apply the join optimization method during the creation of the cartesian product. And that will filter only the joins that can return some result. And because we do it on the fly we discard lots of joins before they are completed. Most of them will be discarded when it creates the join of two basic graph patterns with a shared variable, because typically the

shared variables are in the subject of the pattern and the IRIs (usually mapped using a template). And they will not match so often (commonly the subject IRI template uniquely identifies the R2RML triples map.

The merge with the join optimization can be even more performant if we sort the joins (using the associativity and commutativity, theorem 3.19) in a way that the basic graph patterns with shared variables will be as close as possible, so the join optimization will be able to discard the joins sooner (and therefore it will work faster).

5.1.4 The select into union optimization

Using the method mentioned in subsection 4.4.7 it is possible that the column will rise dramatically. For example, if we have 10 sources and each will need 10 columns for the value binders, then we can create the union operator with 101 columns (and most of the values will be a null value). In most of the cases, it is unnecessary, especially when there is a select over the union operator that will reduce the variable count.

We can decide which variables will be needed over the union operator and then apply simple projection operator over the operands of the union operator. That will possibly reduce the variable count (and so it can also reduce the column count).

5.2 SQL query optimization

In this chapter, we will discuss several methods that can be used to optimize the SQL query that is generated using the proposed transformation.

5.2.1 The condition optimization

The query transformation can create quite complex conditions that are more complex than it is necessary. In this subsection, we will show methods that can be used to optimize these conditions.

We simplify the conditions and also in some cases we detect the cases when we can decide the condition result without the knowledge of data (the condition that will always be false or always true). And we work with these conditions through the logical operands. From the `AND` (alternatively `OR`) operator we can remove all operands that we know that are always true (alternatively always false). And if we find one operand in the `AND` (alternatively `OR`) operator that is always false (alternatively always true) we can then replace the whole operator with a condition that is always false(alternatively always true). But it does not need to be the same condition that we marked as always false(alternatively always true), that condition can be still quite complex for the SQL query engine, so we will replace it with a condition in a simple form, `1=0` (alternatively `1=1`).

The null column optimizer

In the produced SQL query, there will be lots of conditions that will detect whether the column value is NULL or not. Even a simple join of two triple patterns can result in a query with several "is NULL" conditions (as in the figure 5.5).

```
1 SELECT COALESCE(L1.Code, L2.Code) AS Code
2 FROM (SELECT Code FROM Lau WHERE Code IS NOT NULL) AS L1
3 INNER JOIN (SELECT Code FROM Lau WHERE Code IS NOT NULL) AS L2
4 ON (L1.Code IS NULL) OR (L2.Code IS NULL) OR (L1.Code = L2.Code)
```

Figure 5.5: Sample SQL query for join operator

To improve this query we need to analyze whether the column can be NULL or not. We can detect it from the source that is the origin of the column, from conditions that are applied or from the expression that is used to create the column. For example, if we use the query from 5.5 as a subquery of an another query and we want to decide whether the column `Code` can be NULL. It is NULL if and only if `COALESCE(L1.Code, L2.Code)` is NULL, that means if `L1.Code` and `L2.Code` are both NULL. When we check their sources, we can see that none of them can be NULL, so the column `Code` can be NULL.

Even if there will not be the condition `WHERE Code IS NOT NULL`, we can load the database schema for the `Lau` table to get the information whether the column can be null.

If we get the information that the column cannot be NULL, we will then replace the "is NULL" conditions with some condition that is always false. We can also transform some expression, for example, the `COALESCE` expression can remove all arguments after first that cannot be NULL. And in the case that there will remain only one argument, we can replace the `COALESCE` expression with the argument. So the sample from the figure 5.5 can be transformed into the form shown in the figure 5.6 (5.6a is the case when we do not have any information from the database schema, or we have the information that the column `Code` can be null, 5.6b is the case when we get from database schema that the column `Code` can not be null.

```
1 SELECT L1.Code AS Code
2 FROM (SELECT Code FROM Lau WHERE Code IS NOT NULL) AS L1
3 INNER JOIN (SELECT Code FROM Lau WHERE Code IS NOT NULL) AS L2
4 ON L1.Code = L2.Code
```

(a) Without information from the database schema

```
1 SELECT L1.Code AS Code
2 FROM (SELECT Code FROM Lau) AS L1
3 INNER JOIN (SELECT Code FROM) AS L2
4 ON L1.Code = L2.Code
```

(b) With the information from the database schema that `Code` cannot be NULL

Figure 5.6: Sample transformed SQL query for join operator

The concatenation optimization

When we are creating conditions for the value binders that are based on the mapping containing the template, the conditions will usually be in the form of comparison of the `CONCAT` expression. But this condition can be possibly split up. We can compare prefixes and suffixes and if it is an IRI safe values from columns we can split the concatenation more precisely (it is the same logic as in the filtering of the restricted basic graph pattern, subsection 5.1.1).

The splitting of the concatenation works, in a way, as seen in the figure 5.7. The first step is to split up the concatenation. And then we can remove the conditions that are decidable whether they will always be true (comparison of two constants). If there is some that will always be false then the whole condition is always false.

```
CONCAT('http://s.com/', L.Code, '/', L.Label)
  = 'http://s.com/12/45'
```

(a) Original condition

```
('http://s.com/' = 'http://s.com/')
AND (L.Code = '12')
AND ('/' = '/')
AND (L.Label = '45')
```

(b) Splitted up condition

```
(L.Code = '12') AND (L.Label = '45')
```

(c) Cleaned condition

Figure 5.7: The splitting of the concatenation

Comparison of constants

It was used in the concatenation optimization (in the subsection 5.2.1). However, we will process all comparison of constants, not only the ones that come from the concatenation splitting. For the SQL engine, it is better to compare `1=0` than comparing two long strings as a sample.

5.2.2 Flattening the query

Another possible optimization is the effort to achieve the maximum flatness of the query (having the subquery depth as minimal as possible). We already described the transformation algorithm, in a way, that it will try to produce the flat queries, if possible.

The flat queries are better for the SQL engine, because most of the modern engines contains some form of the query execution optimization. And having deeply nested subqueries is a way to confuse the engine optimization to not be able to work properly.

The query from the figure 5.5 will be generated, in a way as shown in the figure 5.8a. So the optimized version from the figure 5.6b will look like as in the figure 5.8b.

```
SELECT COALESCE(L1.Code, L2.Code) AS Code
FROM Lau AS L1
INNER JOIN Lau AS L2
ON (L1.Code IS NULL) OR (L2.Code IS NULL) OR (L1.Code = L2.Code)
WHERE L1.Code IS NOT NULL AND L2.Code IS NOT NULL
```

(a) Original query flattened

```
SELECT L1.Code AS Code
FROM Lau AS L1
INNER Lau AS L2
ON L1.Code = L2.Code
```

(b) The optimized query flattened

Figure 5.8: Flattening the query

5.2.3 Reduced optimizer

If we have an SQL source that does not need to return every result (not only in the DISTINCT or REDUCED query, but also the exists subquery and so on) we can use another optimization. In the R2RML mapping, there are quite often mappings that do have a constant value (typical sample is the class of the triple map and the predicates). And when the SPARQL query asks for these values and the count is not important for us, we can ask only whether there is at least one (we cannot remove the query, because we still need to check whether there will be at least one instance of the constant value).

The benefit of this optimization is that the query execution will not return as many results. That will make a difference especially with the DISTINCT query because we reduce the count how many times will be the discount processed by the SQL engine (because we know the result of the selected subquery).

Is is quite simple to use this optimization method. We only take subqueries that do not have any non-constant columns (the constant expression as a column does not bother us), and we add a limit to them, to restrict the result count to one.

5.2.4 The union creation optimization

In the subsection 5.1.4, we have shown the problem with the column count when generating the SQL query for the union operator. And we proposed an SPARQL optimization.

We can further reduce the column count if we decide which columns (from different sources) can have the same name across the union (that means to decide which columns can be unioned into one column). For this, we need to analyze the column types (because in the SQL only columns of the same type can be unioned).

5.3 Other methods

The mentioned methods are used in the transformation algorithm. But there are also other methods that need to be considered. And they should be handled by the user.

The database design can affect the effectiveness of the created query. Ideally we want to query the tables directly but SQL (and also R2RML) language offers us the possibility to create views. For example the sample from the figure 1.8 can be changed to take a view as a logical table. For example, in a form as seen in the figure 5.9. It represents the exact RDF values, but the transformation algorithm will loose the information that the subject is identified by the `Code` column.

```
1 SELECT
2   CONCAT('http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/',
       ↪ Code) AS Id,
3   Name,
4   Level,
5   CONCAT('http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/',
       ↪ ParentRegion) AS ParentId
6   FROM Nuts
```

Figure 5.9: Sample view for RDF data

Even when we produce simple query from such view, we can notice the performance difference. Although the query from the figure 5.10a seems to be simpler and more efficient, it is slower than the query from the figure 5.10b (in our testing

environment that produced 500 results it was 15% slower).

```
SELECT N.Id AS Id, N2.Id AS ParentId
FROM NutsView AS N
INNER JOIN NutsView AS N2 ON N2.Id = N.ParentId
```

(a) Using a view

```
SELECT
  CONCAT('http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/',
         N.Code) AS Id,
  CONCAT('http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/',
         N2.Code) AS ParentId
FROM Nuts AS N
INNER JOIN Nuts AS N2 ON N2.Code = N.ParentRegion
```

(b) Without a view

Figure 5.10: SQL query with view and without a view

In the common queries, there will the most of the conditions use the columns that are used to create the subject. Because these columns will be usually used in the joins from the basic graph patterns. So for the database design is very crucial that these queries should be as effective as possible. That means that it is ideal to have the subject created from columns (preferable a single column) that can be easily compared (for example numeric types).

It is good to use indexes, to improve the effectiveness of the column comparison. However, we need to take into consideration, that over-indexing can result into a worse performance than under-indexing. The proper index selection will depend on the database schema, and the typical queries used over the database. However, in most of the cases, it will be good to have indexes over the columns that are used as a subject in some triple map.

Although the typical usage of the R2RML mapping is the case when we have a relational database, and we create the R2RML mapping over the fixed database schema, there is another thing to take into consideration. In the sample 1.1, we have shown the relational database schema for the NUTS regions. There are 4 levels of the NUTS regions, and it is possible that every level can have some extra properties. This is the case of type inheritance, and there are several ways how to represent it in a relational database. Two main possible approaches are:

- One table containing the share properties and several tables with the con-

crete types

- One large table containing all possible properties (with NULL values when the property is not relevant) and a column defining the instance type (actually the approach used in the sample 1.1, the `Level` column determines the type)

We will not discuss their advantages and disadvantages in the term of the relational database storage. If the inherited types do have so different subjects that we cannot write the shared type into a shared triples map and we need to create special logical views for the inherited types. But commonly the inherited types will have very similar subject so it can be possible to create one triples map with all shared properties (or with all properties in the second approach).

But what about shared properties defined in different triple maps. For example, if we have the same properties, although they are created in other way for every type. As a sample, we can think about the case with five inherited types and four properties that will be in all inherited types with the same name but specific value creation. Then the SPARQL query asking for all these properties will result in the union of 625 (5^4) joins. So it is good to avoid this scenario, because in larger queries the complexity can grow exponential.

6. Evaluation

In this chapter, we will evaluate our transformation algorithm. First we will show several tests that are used to prove the correctness. Then we will compare the performance of our tool with other possible approaches.

We are not able to evaluate the whole scope of the algorithm. We can test only the part that is implemented. Also, the evaluation lacks from the current state of the implementation. If there is some bad result, we will discuss the reason and what can be done in the future to improve it.

The actual R2RML mapping and database data can be found on the attached CD (details described in appendix A).

6.1 Correctness

The current implementation is able to serve the queries containing only the BGP pattern (with limited support for the property path). The queries we will check are shown on the figures 6.1, 6.6, 6.8, 6.10 and 6.12.

```
1 PREFIX schema:  <http://schema.org/>
2
3 SELECT *
4 WHERE
5 {
6   ?law a schema:Law
7 }
```

Figure 6.1: Query #1

The first query only gets the list of all subjects of the law type. This type is represented by a single table (the `Laws` table, see the figure 6.2) with 30 rows. The query is translated into the SQL query 6.4. It makes a simple SELECT to the `Laws` table and filters out the rows where the column `LawId` is null, because that is the column needed to build the subject of the triples. It is a part of the template in the R2RML mapping file. The SQL query is then successfully converted into the SPARQL result form, and it contains 30 results.

LawId	Law
991	OpatřMZDR
992	Syst.RAPEX
993	Vyhl.174/1992
994	Vyhl.478/2000
995	Zák. 102/2001
996	Zák. 145/2010
997	Zák. 159/1999
998	Zák. 185/2001
999	Zák. 22/1997
1000	Zák. 226/2013
1001	Zák. 247/2006
1002	Zák. 253/2008
1003	Zák. 255/2012
1004	Zák. 256/2001
1005	Zák. 307/2013
1006	Zák. 311/2006
1007	Zák. 321/2001
1008	Zák. 353/2003
1009	Zák. 379/2005
1010	Zák. 455/1991
1011	Zák. 477/2001
1012	Zák. 500/2004
1013	Zák. 539/1992
1014	Zák. 552/1991
1015	Zák. 56/2001
1016	Zák. 634/1992
1017	Zák. 64/1986
1018	Zák. 676/2004
1019	Zák. 86/2002
1020	Zák. OSTATNÍ

Figure 6.2: The Laws table

#	?ca
1	http://linked.opendata.cz/resource/domain/coi.cz/law/991
2	http://linked.opendata.cz/resource/domain/coi.cz/law/992
3	http://linked.opendata.cz/resource/domain/coi.cz/law/993
4	http://linked.opendata.cz/resource/domain/coi.cz/law/994
5	http://linked.opendata.cz/resource/domain/coi.cz/law/995
6	http://linked.opendata.cz/resource/domain/coi.cz/law/996
7	http://linked.opendata.cz/resource/domain/coi.cz/law/997
8	http://linked.opendata.cz/resource/domain/coi.cz/law/998
9	http://linked.opendata.cz/resource/domain/coi.cz/law/999
10	http://linked.opendata.cz/resource/domain/coi.cz/law/1000
11	http://linked.opendata.cz/resource/domain/coi.cz/law/1001
12	http://linked.opendata.cz/resource/domain/coi.cz/law/1002
13	http://linked.opendata.cz/resource/domain/coi.cz/law/1003
14	http://linked.opendata.cz/resource/domain/coi.cz/law/1004
15	http://linked.opendata.cz/resource/domain/coi.cz/law/1005
16	http://linked.opendata.cz/resource/domain/coi.cz/law/1006
17	http://linked.opendata.cz/resource/domain/coi.cz/law/1007
18	http://linked.opendata.cz/resource/domain/coi.cz/law/1008
19	http://linked.opendata.cz/resource/domain/coi.cz/law/1009
20	http://linked.opendata.cz/resource/domain/coi.cz/law/1010
21	http://linked.opendata.cz/resource/domain/coi.cz/law/1011
22	http://linked.opendata.cz/resource/domain/coi.cz/law/1012
23	http://linked.opendata.cz/resource/domain/coi.cz/law/1013
24	http://linked.opendata.cz/resource/domain/coi.cz/law/1014
25	http://linked.opendata.cz/resource/domain/coi.cz/law/1015
26	http://linked.opendata.cz/resource/domain/coi.cz/law/1016
27	http://linked.opendata.cz/resource/domain/coi.cz/law/1017
28	http://linked.opendata.cz/resource/domain/coi.cz/law/1018
29	http://linked.opendata.cz/resource/domain/coi.cz/law/1019
30	http://linked.opendata.cz/resource/domain/coi.cz/law/1020

Figure 6.3: The SPARQL result for the query 6.1

```
SELECT tab.LawId AS LawId
FROM dbo.Laws AS tab
WHERE NOT tab.LawId IS NULL
```

Figure 6.4: SQL query for query 6.1

For this particular query, we will show the source and the final data. The source database table is shown on the figure 6.2). The R2RML mapping of the subject for the triple map is a template, as it is seen in the whole R2RML `LawTriples` mapping (in the figure 6.5). And as can be seen on the result (the figure 6.3) the translation into the SPARQL result form works correctly.

Although it is a simple SPARQL query, it has to compare the predicate and the object. It can be (and it is) done statically, so we exactly know where to get the triples.

```
<LawTriples> a rr:TriplesMap;
  rr:logicalTable [ rr:tableName "[dbo].[Laws]" ];
  rr:subjectMap [
    rr:template "http://linked.opendata.cz/resource/domain/coi.cz/law/{LawId}";
    rr:class schema:Law;
  ];
  rr:predicateObjectMap [
    rr:predicate rdfs:label;
    rr:objectMap [ rr:column "[Law]"; ];
  ];
.
```

Figure 6.5: The R2RML mapping for laws

To test the filtering that cannot be done statically, there is the second query (the figure 6.6). For the simplicity, we use very similar query, but we add a triples pattern that will request another triple with the same subject, the predicate `rdfs:label` and the literal object `Syst.RAPEX`.

The second query translates into the SQL query in the figure 6.7. It is very similar to the query from the figure 6.7, but it joins the same table to itself and additionally filters out the results according to the `Law` column. The query efficiency will be discussed in the following section, where we evaluates the query performance. There is also discussed the presence of the casts, that are not needed in this query. The SQL query returns a single result. The value `992` and the created SPARQL result is also only one, with the value `http://linked.opendata.cz/resource/domain/coi.cz/law/992` assigned to the `ca` variable.

```
PREFIX schema:  <http://schema.org/>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>

SELECT *
WHERE
{
  ?ca a schema:Law;
    rdfs:label "Syst.RAPEX"
}
```

Figure 6.6: Query #2

```
SELECT tab.LawId AS LawId
FROM [dbo].[Laws] AS tab
INNER JOIN [dbo].[Laws] AS tab2
  ON CAST(tab.LawId AS nvarchar(MAX))=CAST(tab2.LawId AS nvarchar(
     ↪ MAX))
WHERE
  NOT tab.LawId IS NULL
  AND NOT tab2.LawId IS NULL
  AND CAST(tab2.[Law] AS nvarchar(MAX))=CAST('Syst.RAPEX' AS
     ↪ nvarchar(MAX))
```

Figure 6.7: SQL query for query 6.6

The third query is getting more complex so we will not show the R2RML mapping. The `schema:CheckAction` represents an inspection that can end (`schema:result`) in a ban or sanction. The sanction (not a ban) has a financial impact (`schema:result`) on the inspected. And the financial value is stored in `gr:hasCurrencyValue`. There are 86009 check actions, 26526 sanctions and every sanction has set the financial value. However, 2215 sanctions refer to the check actions that are not stored in our database (they are in the previous time period that is not present in our database). So there are 24311 sanctions with assigned financial value and which check action is present in our dataset. The SPARQL query is converted into the SQL query from the figure 6.9. It has the same issues as the previous one (and it will be also discussed in the following section), but it correctly returns 24311 rows that are correctly transformed into the SPARQL result.

```
PREFIX schema:  <http://schema.org/>
PREFIX gr:      <http://purl.org/goodrelations/v1#>

SELECT *
WHERE
{
  ?ca a schema:CheckAction;
    schema:result/schema:result/gr:hasCurrencyValue ?result.
}
```

Figure 6.8: Query #3

```
SELECT tab6.[Sanction] AS [Sanction], tab.Id AS Id
FROM [dbo].[CheckAction] AS tab
INNER JOIN [dbo].[CheckAction] AS tab2
  ON CAST(tab.Id AS nvarchar(MAX))=CAST(tab2.Id AS nvarchar(MAX))
INNER JOIN [dbo].[Sanction] AS tab3
  ON CAST(tab2.[Id] AS nvarchar(MAX))=CAST(tab3.[CheckActionId] AS
     ↪ nvarchar(MAX))
INNER JOIN [dbo].[Sanction] AS tab4
  ON CAST(tab3.Id AS nvarchar(MAX))=CAST(tab4.Id AS nvarchar(MAX))
INNER JOIN [dbo].[Sanction] AS tab5
  ON CAST(tab4.[Id] AS nvarchar(MAX))=CAST(tab5.[Id] AS nvarchar(
     ↪ MAX))
INNER JOIN [dbo].[Sanction] AS tab6
  ON CAST(tab5.Id AS nvarchar(MAX))=CAST(tab6.Id AS nvarchar(MAX))
WHERE NOT tab.Id IS NULL AND NOT tab2.Id IS NULL
  AND NOT tab3.Id IS NULL AND NOT tab4.Id IS NULL
  AND NOT tab5.Id IS NULL AND NOT tab6.Id IS NULL
  AND NOT tab6.[Sanction] IS NULL
```

Figure 6.9: SQL query for query 6.8

The fourth query contains the `DISTINCT` query of all classes present in the query. Although the R2RML mappping has set all the classes statically, we need to perform a query to decide whether there is at least one instance or not. So the final SQL query will look as in the figure 6.11. It correctly returns 14 results. That means all classes defined in the mapping because all the tables used in triple maps are not empty and produce at least one triple.

```
SELECT DISTINCT ?t
WHERE
{
  [] a ?t
}
```

Figure 6.10: Query #4

```
SELECT DISTINCT
CASE
  WHEN CAST(un.uncase AS nvarchar(MAX))=CAST(0 AS nvarchar(MAX))
    THEN 'http://ec.europa.eu/eurostat/ramon/ontologies/geographic
       ↪ .rdf#NUTSRegion'
  WHEN CAST(un.uncase AS nvarchar(MAX))=CAST(1 AS nvarchar(MAX))
    THEN 'http://ec.europa.eu/eurostat/ramon/ontologies/geographic
       ↪ .rdf#LAURegion'

  ...
END AS expr
FROM (
  SELECT TOP 1 0 AS uncase
  FROM [dbo].[Nuts] AS tab
  WHERE NOT tab.Code IS NULL

  UNION

  SELECT TOP 1 1 AS uncase
  FROM [dbo].[Lau] AS tab2
  WHERE NOT tab2.Code IS NULL

  UNION

  ...
)
```

Figure 6.11: SQL query for query 6.10

The last fifth query only extends the previous one, but instead of the DISTINCT we use the REDUCED. It joins the classes with their found properties. Now it is not needed to convert the value binders, so we do not have to convert the value binders into the exact value. So the SQL query is a list of numbers and each of it represents one combination of class and property. The query correctly returns all 73 combinations.

```
SELECT REDUCED ?class ?prop
WHERE
{
  [] a ?class;
    ?prop [].
}
```

Figure 6.12: Query #5

```
SELECT DISTINCT un.uncase AS expr FROM (
  SELECT TOP 1 0 AS uncase FROM [dbo].[Nuts] AS tab
  INNER JOIN [dbo].[Nuts] AS tab2
  ON CAST(tab.Code AS nvarchar(MAX))=CAST(tab2.Code AS nvarchar(
     ↪ MAX))
  WHERE NOT tab.Code IS NULL AND NOT tab2.Code IS NULL AND NOT
     ↪ tab2.Level IS NULL

  UNION

  ...
) AS un
```

Figure 6.13: SQL query for query 6.12

6.2 Performance

We evaluate the performance of the implementations using the created SPARQL endpoints. We use develop branch of the D2RQ implementation[1], the Virtuoso Universal Server and our implementation. All these contain the support for creating the standard SPARQL endpoint. To create the proper environment we test it in a virtual machine hosted on a PC with Intel Core 2 Duo P9500 CPU[2]. The virtual machine has assigned 4GB of RAM (from a total of 8GB on the host PC). As the back database was used the MS SQL 2014 Express, the D2RQ code is latest[3] and the Virtuoso Universal Server is in version 07.10.3207. The virtual machine runs Windows 7 64-bit.

[1]The D2RQ is missing several entries, because the queries was not served by the endpoint. The implementation of the R2RML in the D2RQ server is not a final version, so we can test only a minority of queries.

[2]2 cores clocked on 2.53GHz

[3]Downloaded June, 2014

Query #	Our	D2RQ	Virtuoso
1	11.5ms	24.8ms	3.24ms
2	18.0ms	-	2.97ms
3	2562.1ms	-	603.6ms
4	12.0ms	2763.6ms	34.9ms
5	267.6ms	-	4866.0ms

Figure 6.14: Table of query execution times

We use the same queries as in the correctness evaluation (figures 6.1, 6.6, 6.8, 6.10 and 6.12). Every query was executed 15-times against the created endpoint. Then the best and the worsest time was removed and in the table in the figure 6.14 (or comparison in the figure 6.15) you can see the average of the rest 13 executions.

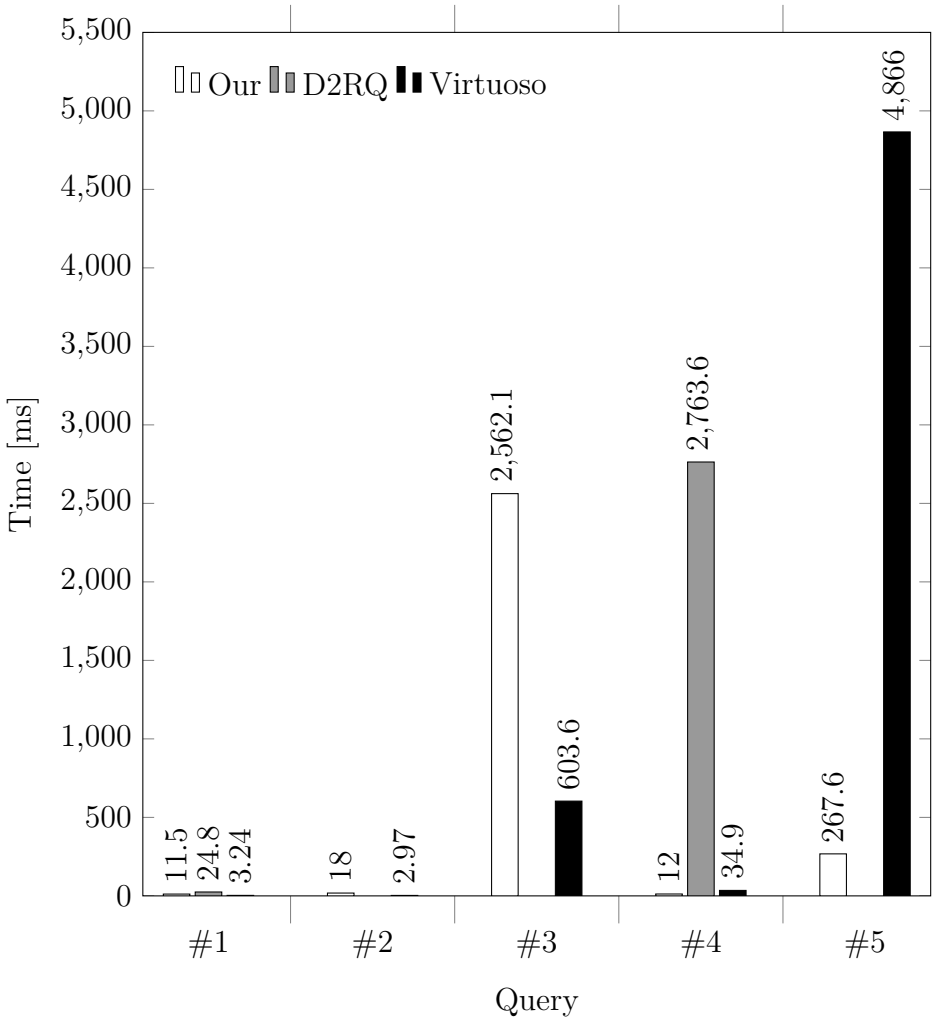


Figure 6.15: Query execution times comparison

As you can see our transformation works pretty well for the queries #1, #2, #4 and #5. Although the Virtuoso server is faster for the queries #1 and #2, there is not so much we can do more to optimize the results. The native solution is just faster for this type of queries, and we will hardly beat that. On the other hand, the times are so low, that it probably does not matter.

For the queries #4 and #5 our solution provides the best result. That is because we need to do only a minimal query to the database, we know the result from the mapping file. It seems like the D2RQ server does not do optimizations like our solution. Interesting is the difference between the query execution time of the queries #4 and #5 using the Virtuoso endpoint. It seems like the classes are indexed, but their properties do not have an index. So the query perform very badly.

The query #3 performs very badly using our solution. We believe that it can be improved. And probably provide very similar performance as the Virtuoso or maybe even slightly better. The issue is in the generated SQL query (the figure 6.9). There are present too many unneeded casts. It is because our implementation is not yet able to load the SQL types from the database. If we implement a database schema loader, that will get all the information about the database schema we will be able to remove the casts from the query (so the query will look as it is in the figure 6.16).

```
SELECT tab6.[Sanction] AS [Sanction], tab.Id AS Id
FROM [dbo].[CheckAction] AS tab
INNER JOIN [dbo].[CheckAction] AS tab2
  ON tab.Id=tab2.Id
INNER JOIN [dbo].[Sanction] AS tab3
  ON tab2.[Id]=tab3.[CheckActionId]
INNER JOIN [dbo].[Sanction] AS tab4
  ON tab3.Id=tab4.Id
INNER JOIN [dbo].[Sanction] AS tab5
  ON tab4.[Id]=tab5.[Id]
INNER JOIN [dbo].[Sanction] AS tab6
  ON tab5.Id=tab6.Id
WHERE NOT tab.Id IS NULL AND NOT tab2.Id IS NULL
  AND NOT tab3.Id IS NULL AND NOT tab4.Id IS NULL
  AND NOT tab5.Id IS NULL AND NOT tab6.Id IS NULL
  AND NOT tab6.[Sanction] IS NULL
```

Figure 6.16: Optimized version of the query 6.9

And we can optimize it even more. If we load the information that the columns `Id` of the table `CheckAction` and `Sanction` are their primary keys (we only need

the uniqueness) we can remove the joins (because we know, that we cannot join the row with another row than with the same one). So we will get even better query, from the figure 6.17.

```
1 SELECT tab2.[Sanction] AS [Sanction], tab.Id AS Id
2 FROM [dbo].[CheckAction] AS tab
3 INNER JOIN [dbo].[Sanction] AS tab2
4   ON tab.[Id]=tab2.[CheckActionId]
5 WHERE NOT tab.Id IS NULL AND NOT tab2.Id IS NULL
6   AND NOT tab2.[Sanction] IS NULL
```

Figure 6.17: Optimized version of the query 6.9

The SQL query 6.17 executes approximately ten times faster than the query 6.9, so we believe that it will be possible to reduce the execution time. It will not be ten times faster because the SQL query execution time is not the only factor. For example, it will take more time to create an optimized query like this, and there are things that will take the same time as before, like final transformation of the SQL results. However, we believe that it is possible to have very similar or even better time than the Virtuoso server.

6.3 Payola

The target of the implementation is to support the queries for an analysis in the payola system. The selected analysis is "COI.CZ inspections and sanctions by regions and sanction value"[4]. The required queries are shown in the figures 6.18 and 6.19.

We will not show the transformed queries because they are too long. They do have the same performance issue as mentioned in the previous section for the query #3 (in the figure 6.8). So we believe that we will be able to create a more performant transformation using the improvement mentioned in the previous section. However, there is also another thing that results in the less performant query. The check action location contains a reference to the LAU regions of both levels using the same predicate `s:location`. Although we know which node refers which level, the transformation algorithm does not have this

[4] Available at `http://live.payola.cz/analysis/7b2ee8cc-f03a-4a04-ba9d-e54e65346191` (visited July, 2014)

```
PREFIX gr: <http://purl.org/goodrelations/v1#>
PREFIX ruian: <http://ruian.linked.opendata.cz/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ec: <http://ec.europa.eu/eurostat/ramon/ontologies/
   ↪ geographic.rdf#>
PREFIX s: <http://schema.org/>
PREFIX dcterms: <http://purl.org/dc/terms/>

CONSTRUCT {
  ?ca a s:CheckAction;
  s:location ?region ;
  s:geo ?geo;
  s:title ?title;
  s:description ?desc;
  dcterms:date ?date ;
  rdf:value ?value.
  ?geo s:latitude ?lat;
  s:longitude ?lon.
}
WHERE
{
  ?ca a s:CheckAction;
  s:location/s:location ?region;
  s:location/s:geo ?geo;
  s:object ?object;
  dcterms:date ?date ;
  s:result ?result.
  ?result a <http://linked.opendata.cz/ontology/coi.cz/Sanction>;
  s:result/gr:hasCurrencyValue ?value.
  ?object gr:legalName ?title .
  ?region a ec:LAURegion;
  ec:level 2.
  ?geo s:latitude ?lat;
  s:longitude ?lon.
  BIND (CONCAT('<a href="', ?object,'">', ?title, '</a>') as ?desc
     ↪ )
}
```

Figure 6.18: Query #1 for Payola analysis

information because both levels have the same subject map. So it is not possible to decide which node refers to the LAU region of `ec:level` 2.

The performance is shown in the figure 6.20. It was tested on the same hardware as used in the performance section.

Although the performance is not yet able to compete with the Virtuoso Universal Server, it returns correct results. For the query #1, the where condition matches 4273 solution mappings. They are then transformed into 37572 unique triples. The construct template has 9 triple patterns, so the expected count is 38457. The difference is 885 triples. That is because the returned solution

```
PREFIX gr: <http://purl.org/goodrelations/v1#>
PREFIX ruian: <http://ruian.linked.opendata.cz/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ec: <http://ec.europa.eu/eurostat/ramon/ontologies/
   ↪ geographic.rdf#>
PREFIX s: <http://schema.org/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

CONSTRUCT {
  ?region a ec:LAURegion;
  ec:level 2;
  rdfs:label ?l2label ;
  ec:hasParentRegion ?lau1.
  ?lau1 rdfs:label ?l1label ;
  ec:hasParentRegion ?nuts3 .
  ?nuts3 rdfs:label ?n3label ;
  ec:hasParentRegion ?nuts2 .
  ?nuts2 rdfs:label ?n2label ;
  ec:hasParentRegion ?nuts1 .
  ?nuts1 rdfs:label ?n1label.
}
WHERE
{
  ?check a s:CheckAction;
  s:location/s:location ?region;
  s:result/s:result [] .
  ?region a ec:LAURegion;
  ec:level 2;
  dcterms:title ?l2label ;
  ec:hasParentRegion ?lau1.
  ?lau1 dcterms:title ?l1label ;
  ec:hasParentRegion ?nuts3 .
  ?nuts3 rdfs:label ?n3label ;
  ec:hasParentRegion ?nuts2 .
  ?nuts2 rdfs:label ?n2label ;
  ec:hasParentRegion ?nuts1 .
  ?nuts1 rdfs:label ?n1label.
}
```

Figure 6.19: Query #2 for Payola analysis

Query #	Our	Virtuoso
1	6098.7ms	1418.1ms
2	8492.4ms	2021.9ms

Figure 6.20: Performance of the Payola queries

mappings contain 4164 unique check actions. So, without the triple with predicate `rdf:value`, we generate 33312 triples. The solution mappings contain 4260 unique combinations of check action and sanction value. So the total count should be 37572 triples, and that is the returned count of triples.

For the query #2, the where condition matches 5217 solution mappings. They contain 934 unique LAU2 regions, 75 unique LAU1 regions, 13 unique NUTS3 regions, 7 unique NUTS2 regions and 1 unique NUTS1 region. That means $934 * 4 + 75 * 2 + 13 * 2 + 7 * 2 + 1 = 3927$ unique triples. That is exactly the returned count of triples.

7. Implementation

One of the main purposes of this thesis is also the implementation of the tool that can be used as a virtual SPARQL endpoint over the relational data. As we mentioned in the evaluation chapter 6, we did not implement the full support for the SPARQL algebra.

The tool (called R2RMLStore) basically offers the support for:

- The triple pattern (containing variables, nodes or blank nodes)
- Join (multiple triple patterns)
- Property path (but only sequence, inverse and alternative path)
- Limit and offset (but offset can be used only when ordering is used)
- SELECT clause
- CONSTRUCT clause
- BIND pattern (but the expression can contain only CONCAT)
- REDUCED operator

There is also a limited support for:

- DISTINCT
- Ordering

The limited support for these two types means that it is partly implemented, but it needs some extra work. The DISTINCT operator loses all information about the node types (whether it is an IRI, literal and the type of the literal) and returns everything as a string literal. The implemented ordering does not exactly follow the SPARQL specification, it orders the results only by sorting the string form of the value. But the ordering is needed when we want to use the offset (that is the limitation of the MS SQL RDBMS) so we provide this partial implementation.

7.1 Used technologies

The C# 4.5 (currently the newest version of the major language for the .NET Framework) was selected as the programming language. It is a modern language that enables using of various paradigms – imperative, declarative, generic, object-oriented and event-driven programming. It offers us the option to implement some parts using other .NET languages, like F# that can be more efficient for specific tasks, but the current implementation is purely in C#. The disadvantage of this selection is that the .NET Framework 4.5 runs nowadays only on following operation systems[1]:

- Windows 8.1
- Windows 8
- Windows 7 SP1
- Windows Vista SP2
- Windows Server 2012 R2
- Windows Server 2012 (64-bit edition)
- Windows Server 2008 R2 SP1
- Windows Server 2008 SP2

We have used third-party libraries to work with the RDF data. That means the dotNetRDF library (section 2.5) for working with RDF and SPARQL. And the r2rml4net (section 2.6) library for loading the mapping file. These two are the major libraries for their purposes on the .NET platform.

As a relational database, we have chosen the MS SQL 2014. That is currently the newest version of a standard database used in .NET applications. But we plan to implement the support also for other RDBMS.

To create a website that will host the SPARQL endpoint we have chosen the ASP.NET MVC 5 website. That is a standard approach when building .NET websites and we did not have any requirements for our tool. The website is only hosting an already implemented endpoint (in the dotNetRDF library).

[1] Using Mono project[2] it may be possible to run the application on other systems, but it was not tested

[2] Available at `http://mono-project.com/` (visited June, 2014)

7.2 Project

The project can be opened in a Microsoft Visual Studio 2013 Update 2 where is installed the NuGet Package Manager extension. The project is separated into the two parts, the website and the storage library. The storage library contains the complete logic for the data querying. The website is responsible for creating the instance of the storage with the selected mapping and connection to a database, configuring the standard SPARQL endpoint and handle all query executions by the user (calling the storage and printing out its results).

7.3 The storage library

The storage library contains all the logic and models needed for the data querying. The library consists principally of the following folders:

- `Mapping` - The R2RML mapping holder and several methods to access the mapping
- `Optimization` - The optimization methods, both for SPARQL and SQL algebra
- `Query` - The processor executing the whole algorithm and the context for the query
- `Sparql` - The models for representing the SPARQL algebra and the builder that constructs the model from the dotNetRDF representation
- `Sql` - The models for representing the SQL algebra (including the value binders), vendor implementation (generating the concrete SQL query and executing it) and the builder that creates the SQL algebra from the SPARQL algebra

In the root folder there is the class `R2RMLStorage` that is used from the applications. This class is only a holder implementing a needed dotNetRDF interface (`IQueryableStorage`) and for every query calling a proper method on a `QueryProcessor`.

7.3.1 Processing the query

The processing of the query is shown on the figure 7.1. The query enters the query processor where we create the context (`QueryContext`) and prepare the result handlers. Then the SPARQL query is parsed (using the dotNetRDF library). The parsed query is then transformed (using the class `SparqlAlgebraBuilder`) to our algebra representation (an instance of the interface `ISparqlQuery`). Then is the algebra transformed by adding the R2RML mapping information (using the class `MappingProcessor`). After that, we call every optimizer for the SPARQL query (registered instances of the interface `ISparqlAlgebraOptimizer`). At this point, we have our final SPARQL algebra.

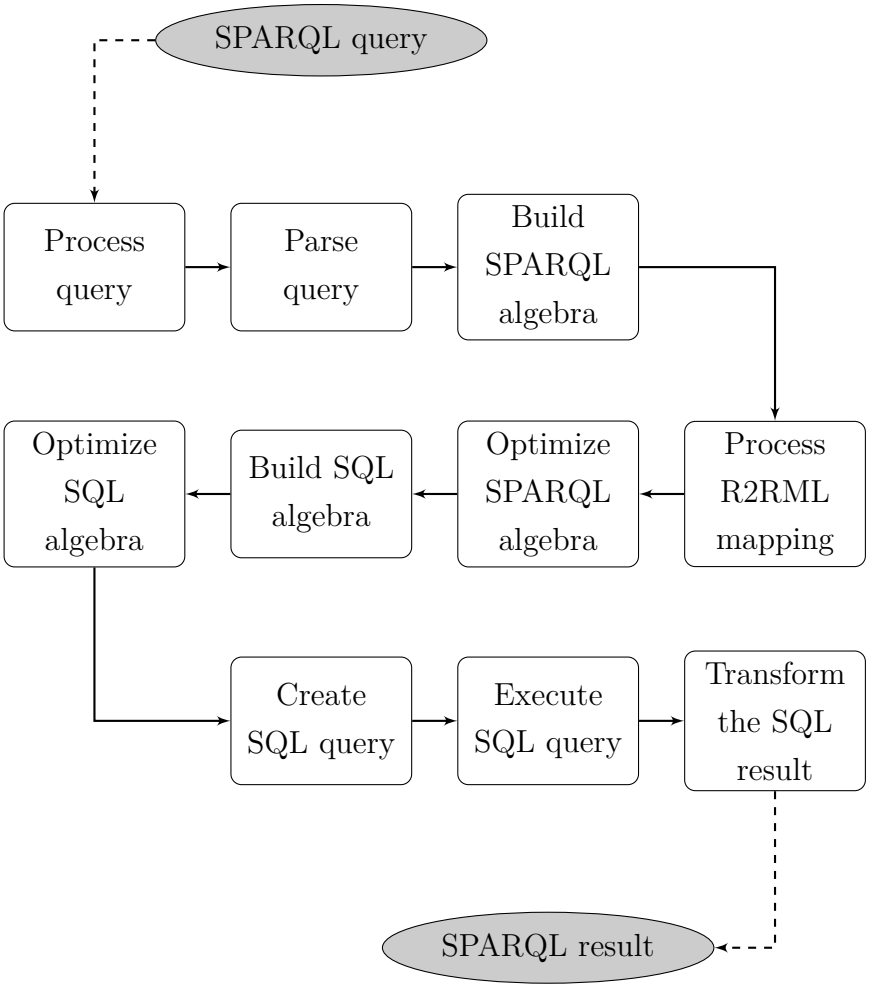


Figure 7.1: Processing the query

The next step is to convert the SPARQL algebra into the SQL form. For that, there is the class `SqlAlgebraBuilder`. During the conversion, there are called on-the-fly optimizers (registered instances of the interface `ISqlAlgebraOptimizer-OnTheFly`) for created parts. The created SQL algebra is an instance of the

interface `INotSqlOriginalDbSource` that represents not only an SQL query, but also value binders that are represented by the query (instances of the interface `IBaseValueBinder`). After that, we optimize the SQL algebra (using registered instances of the interface `ISqlAlgebraOptimizer`).

The created SQL algebra is then converted to the concrete SQL query. First all used columns and sources are named (using the class `BaseSqlNameGenerator`) and then the query is created (using the class `BaseSqlQueryBuilder`). We are preparing the implementation to the form where the conversion will be handled according to the selected database vendor.

Then the `QueryProcessor` calls the database vendor implementation (the class `MSSQLDb`) to execute the query. To be able to work with different databases, we have our special interface `IQueryResultReader` that represents the reader of the query execution results. So for other vendors than MS SQL we can use some absolutely different approach to the query execution than the standard ADO.NET.

The last step is to convert the SQL result into the SPARQL form. It differs according to the type of the query. For the select form, we simply convert the values using the value binders (created with the SQL algebra). For other forms, we do extra processing like transformation using the CONSTRUCT clause template.

The value binders create the concrete value using the `LoadNode` method. This method is called for every row, so it is needed to be optimized. Especially for the standard value binder `ValueBinder`, it contains complex logic that depends on the R2RML mapping. For this value binder we prepare the LoadNode function on the first call, we create an expression tree where all conditions depending on the R2RML mapping is already evaluated. This expression tree is compiled into a function, cached (so the function is created only on the first call) and then called.

The instance registration is done in the class `QueryProcessor`, but in the future it will be done using some dependency injection container to make it more flexible and configurable. Also, there is currently no support for configuration or logging.

To implement the support for another SPARQL query you need to modify the `SparqlAlgebraBuilder` class. This class converts the dotNetRDF representation into our algebra representation. If needed, the new SPARQL oper-

ator should be implemented in the `Slp.r2rml4net.Storage.Sparql.Algebra.Operator` namespace and inherit the `ISparqlQueryPart` interface (the `ISparqlQueryModifier` interface for the solution modifiers). The SPARQL expression parts are in the `Slp.r2rml4net.Storage.Sparql.Algebra.Expression` namespace and inherit the `ISparqlQueryExpression`. The new parts need to be supported in the `SqlAlgebraBuilder` class, that converts the SPARQL algebra to the SQL representation. It may be needed to create new classes to represent another statement (the `ISqlOriginalDbSource` interface for original database sources, the `INotSqlOriginalDbSource` interface for created SQL statements that can be used as a source), condition (the `ICondition` interface), expression (the `IExpression` interface) or value binder (the `IBaseValueBinder` interface). That is all enclosed in the `Slp.r2rml4net.Storage.Sql` namespace. Not only the `SqlAlgebraBuilder` class is used to create an SQL query. The class `ConditionBuilder` is used to create the conditions, and the class `ExpressionBuilder` is used to create the SQL expressions. The text representation of the SQL query is then generated using the `BaseSqlQueryBuilder` class. So the new SQL parts have to be supported also in this class.

It is possible to implement new optimization algorithms. It is needed to inherit the interface `ISparqlAlgebraOptimizer` for the optimizations of the SPARQL algebra, the interface `ISqlAlgebraOptimizer` for the optimizations of the SQL algebra and the interface `ISqlAlgebraOptimizerOnTheFly` for the optimizations of the SQL algebra that are used during the creation of the query. The optimization inheriting `ISqlAlgebraOptimizer` is called only once after the whole algebra was created. The new optimization class needs to be added to the corresponding list in the constructor of the `QueryProcessor` class to register the optimization.

7.4 The website

The website is a standard ASP.NET MVC application. That implies we are using the Model-View-Controller pattern. That means that every request calls a method on a controller, which prepares the data in the form of a model, and the model is then send back as a response in the form that is defined by the view.

The website application is partitioned into the following folders:

- `App_Data` - Configuration and mapping files

- `App_Start` - Routes and bundles registration
- `Content` - Static files (stylesheets, javascripts, fonts and images)
- `Controllers` - Controllers that are handling the requests
- `Models` - Models for the controllers
- `R2RML` - Wrappers for the storage library
- `Views` - Views (html templates) for the controllers

The website provides two options how to execute the query. It can execute the query using our page (that uses Ajax to call our controller) and display the result as part of the HTML page. For this approach, we have implemented a method on `JsonController` that serves the query (and also includes the internal execution time - time spend in the storage library). The result is then written on the page using javascript.

The other option is to use the standard dotNetRDF SPARQL endpoint (implemented as an HttpHandler). It only needs the configuration to access the storage. For this purpose, we needed to create a factory (`R2RMLStorageFactoryForQueryHandler`) that returns the storage when we reference it in the configuration file. It returns the singleton instance of the storage that is hosted in the class `StorageWrapper`).

The class `StorageWrapper` ensures the creation of the `R2RMLStorage` instance. On the application start, it loads the database connection string (from the Web.config file) and the mapping file (path to the file is found in the Web.config file). After that it tries to parse the R2RML mapping (using the r2rml4net library) and then create the storage. If anything fails, it sets the `StartException` property, and that is checked on the every action executing of the `MainController`. If it is not null, it redirects to a special error page where is the exception listed.

8. User guide

The R2RMLStore is a web application that is providing a virtual SPARQL endpoint over relational data stored in the MS SQL database. The mapping is defined by an R2RML mapping file.

8.1 Installation

The R2RMLStore is written for Windows-based hostings capable of running ASP.NET MVC 5 applications on the .NET 4.5 Framework. The supported OS are listed in the section 7.1. For the proper use, we will also need a connection to a MS SQL database, local or remote. We do support MS SQL of version 2012 and newer.

To use the application on a local machine it is needed to have installed Internet Information Services with the support for ASP.NET 4.5 applications. The IIS can be installed using the Add or Remove features dialog in the Windows control panel (we need to install the IIS with all the components connected to the ASP.NET). To register the .NET framework 4.5 into IIS, there is a tool called `aspnet_regiis`[1].

Using the Internet Information Services Manager tool (it can also be installed using the Add or Remove features dialog in the Windows control panel) we have to create a website that will point to some selected folder. Into this folder we will copy the application from the CD (see appendix A).

If we want to use a local MS SQL database[2], we need to install it first. We only follow the installation instructions and create a default or named database instance. The configuration (like instance naming, access type etc.) only modifies the connection string that will be used for the web application configuration.

[1] Description at `http://msdn.microsoft.com/en-US/library/k6h9cz8h(v=vs.80).ASPX` (visited June, 2014)

[2] Express (free) version of the MS SQL is available at `http://www.microsoft.com/en-us/server-cloud/products/sql-server-editions/sql-server-express.aspx` (visited July, 2014)

8.2 Configuration

At this point, it is possible to access the web at the virtual address chosen in the previous step. If the installation proceeded correctly, you should see a page labeled R2RMLStore. It may show an error "Application start failed" or the queries will fail (in error). The SPARQL Endpoint will not be accessible at all.

First we need to set the path to the R2RML mapping definition. This path is stored in the `Web.config` file as the value in the `configuration/appSettings` section with the key `r2rmlConfig`.

Then we need to set the connection string to the MS SQL database. It depends[3] on the SQL server we want to use (whether it is local or remote). There is only one connection string, and it is named `r2rmlstoreconnection`, this connection string will be used by the application. The connection string can be found in the `Web.config` file in the `configuration/connectionStrings` section.

If you configure these two values correctly, it will be possible to run the query successfully. To test the application, you can use sample data and sample mappings from the CD (see appendix A).

8.3 Using the application

There are two possible ways how to use the application. Both options are accessible from the home page, using the links "SPARQL Query" and "SPARQL Endpoint".

The "SPARQL Query" is our implementation of the query execution. It is accessible from the web browser. After typing the query into the large textarea, the query is executed by the "Execute" button. If there is any error, it will be displayed on the top of the page. Otherwise the result (and the time needed) will be shown bellow the query.

The "SPARQL Endpoint" is the standard SPARQL Endpoint implemented in

[3]Samples are shown at `http://www.connectionstrings.com/sql-server-2012/` (visited June, 2014) and at `http://msdn.microsoft.com/en-us/library/jj653752(v=vs.110).aspx` (visited June, 2014)

the dotNetRDF library. Its configuration is in the `~/App_Data/rdf_config.ttl` file[4], but it is set to use the same storage as the "SPARQL Query" and it should not be changed. The "SPARQL Endpoint" does not provide us a pleasant user interface, but it can be used by a third-party application. It tries to return the results in the requested format, so it formats the result as HTML in the browser, but as XML when accessed, for example, from PowerShell (that was the way how the evaluation tests were ran). To run a query from an external tool, we can pass the query as the GET parameter named `query` that means in the query string in the URL encoded format.

When using, the user should keep on mind that the application is now intended to use for smaller queries. It is not implemented for creating database dumps and so on. This restriction results from the fact, that the query execution first loads the whole result in the memory and then the result is written to the output. This is the default dotNetRDF behaviour, and it should be probably changed in the future (for queries with extremely large results).

[4]The configuration is documented at `https://bitbucket.org/dotnetrdf/dotnetrdf/wiki/UserGuide/ASP/Creating%20SPARQL%20Endpoints` (visited June, 2014)

9. Conclusion

In this master thesis, we presented a formal model for a SPARQL algebra that can be used to transform the SPARQL query into the SQL query using a user-defined mapping. This model was then successfully used in the implementation of such an algorithm. The implementation does not fully cover the possibilities of the SPARQL query, but it shows the technique how to transform the query to the SQL form and the SQL result back to the SPARQL form.

Although the transformation is not completely implemented, we have shown that it is possible for the RDB2RDF system to be effective and even able to compete with the native solutions. For the query that was not so efficient we have discussed the way how it can be optimized.

We have found several problems with the transformation. We have shown that the SPARQL and SQL languages differs, in a way that we have to do some extra handling before using the similar operator, or we are not able to use it at all. The difference is primarily in the way of evaluation and how does it work with types.

The dotNetRDF library is primarily aimed at the usage where the full dataset is loaded in memory. That results in the fact that the endpoint is not effective to be used to dump the whole dataset because it typically exceeds the memory capabilities. If it is needed to return large results (the whole dump, large count of triples or solution mappings) it may be better to use native RDF solutions (like virtuoso). However, if the typical queries are returning small results (although the whole dataset can be large), then our solution fits great. Moreover, our solution is quite easy to deploy on an existing website. So we can imagine the usage on various e-commerce and blogging engines.

9.1 Future work

Currently, the implementation has only a limited support for the SPARQL algebra. The first task is to finalize the support for the basic parts of the algebra - loading the database scheme, load the database types. The database types allow us to work correctly as it is defined in the R2RML specification (the final RDF literal type may depend on the database type). Moreover, the schema can be

used (as it is mentioned in the section 6.2) to optimize the query.

The next task will be to complete the implementation to support the whole SPARQL algebra. It may result in several minor changes in the algebra, and it is also possible that there will be found a better approach than the mentioned in this work. However, we believe that the proposed algorithm is implementable, although it will need much code (even for the filter pattern there will be much code for the transformation to the corresponding SQL functions). The newly implemented parts may also result in new optimization methods for them.

There is also a group of optimization methods that has not been discussed at all. We may be able to produce more efficient queries if we first load data statistics from the database. Then we can modify the optimization methods accordingly to these data. For example, using the union optimization (described in the subsection 5.1.3) may not be optimal in all cases.

The current implementation of the SPARQL endpoint is now not working as efficient as it could be. We will need to implement the endpoint, in a way, that it will be able efficiently to work with large datasets that exceed the memory capabilities. The default dotNetRDF endpoint, which is used, is created to work with datasets stored in memory. Therefore, it is implemented, in a way that it stores the whole result in the memory and then it is written to the output.

Bibliography

[1] *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation 25 February 2014.
http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225

[2] *SPARQL 1.1 Query Language*. W3C Recommendation 21 March 2013.
http://www.w3.org/TR/2013/REC-sparql11-query-20130321

[3] *R2RML: RDB to RDF Mapping Language*. W3C Recommendation 27 September 2012.
http://www.w3.org/TR/r2rml/

[4] *RDF 1.1 Turtle*. W3C Recommendation 25 February 2014.
http://www.w3.org/TR/2014/REC-turtle-20140225/

[5] *Linked Data - Connect Distributed Data across the Web*. [Cited: June, 2014.]
http://linkeddata.org/

[6] Cyganiak, Richard. *A relational algebra for SPARQL*. Digital Media Systems Laboratory, HP Laboratories Bristol, 2005. HPL-2005-170
http://www.hpl.hp.com/techreports/2005/HPL-2005-170.pdf

[7] Chebotko, Artem - Lu, Shiyong - Fotouhi, Farshad. *Semantics preserving SPARQL-to-SQL translation*. Data & Knowledge Engineering Volume 68, Issue 10, October 2009, Pages 973–1000

[8] Chebotko, Artem - Lu, Shiyong - Jamil, Hasan M. - Fotouhi, Farshad. *Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns*. Technical Report TR-DB-052006-CLJF, May 2006. Revised November 2006

[9] *Transact-SQL Reference (Database Engine)*. Microsoft Developer Network, [Cited: June, 2014.]
http://msdn.microsoft.com/en-us/library/bb510741

[10] Litwin, Paul. *Fundamentals of Relational Database Design*. Microsoft Access 2 Developer's Handbook, Sybex 1994

[11] Bizer, Christian - Heath, Tom - Berners-Lee, Tim. *Linked Data - The Story So Far*. International Journal on Semantic Web and Information Systems, Vol. 5(3), Pages 1-22 (2009)

[12] Bornea, Mihaela A. - Dolby, Julian - Kementsietsidis, Anastasios - Srinivas, Kavitha - Dantressangle, Patrick - Udrea, Octavian - Bhattacharjee, Bishwaranjan. *Building an efficient RDF store over a relational database*. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13). ACM, New York, NY, USA, Pages 121-132.

[13] Priyatna, Freddy - Corcho, Oscar - Sequeda, Juan. *Formalisation and Experiences of R2RML-based SPARQL to SQL query translation using Morph*. Proceedings of the 23rd International Conference on World Wide Web, International World Wide Web Conferences Steering Committee, Pages 479-490, 2014

[14] Harris, Stephen - Shadbolt, Nigel *SPARQL Query Processing with Conventional Relational Database Systems*. Proceedings of the 2005 International Conference on Web Information Systems Engineering, Springer-Verlag, Pages 235-244, 2005

A. CD Contents

The enclosed CD contains this document in a portable document format, published folder of the web application (in the folder `Publish`), the source codes of the implementation (in the folder `Source`) and the software documentation (in the folder `Documentation`). There are two sample scenarios, the CTIA sample and the demography sample. The CTIA sample (in the folder `Sample/CTIA`) is the sample used in this work, to evaluate the correctness and the performance. It was also used for the Payola evaluation. The source of the data is taken from the public database of check actions, sanctions and bans[1]. The other sample, the demography sample, (in the folder `Sample/Demography`) is a sample taken from open data datasets[2]. This sample models the demographic data about the Czech population. It was used in the early stages of the development.

[1] The source data is available at `http://www.coi.cz/cz/spotrebitel/open-data-databaze-kontrol-sankci-a-zakazu/` (visited June, 2014)

[2] Available at `http://opendata.cz/linked-data` (visited June, 2014)

B. List of Figures