SPARQL to SQL query translation using R2RML mappings

A dissertation submitted in partial fulfilment of the requirements for the MSc in Advanced Computing Technologies

by Sebastian Holzschuher

Department of Computer Science and Information Systems

Birkbeck College, University of London

September 2015

Academic Declaration

This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service. I have read and understood the sections on plagiarism in the Programme Handbook and the College website.

The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

This report documents the design and implementation of a SPARQL to SQL query translator which uses R2RML mappings to reverse-engineer a SQL query against the original relational database from an input SPARQL query. The purpose of the application is publishing information stored in a relational database as Linked Data on the Semantic Web. The reasons for taking this approach are highlighted and explained by comparing this option to various other methods for exposing Linked Data on the web. The report defines a clear scope with regards to the SPARQL language and R2RML mapping specifications and details the technical architecture of the application. Key stages of the translation algorithm are explained and several test cases are presented which document and validate the implementation of translation features. The report concludes with highlighting shortcomings and areas for improvement that could be solved or implemented in future releases.

Supervisor: Michael Zakharyaschev

Table of Contents

Table of Figures

List of Tables

1 Introduction

1.1 Semantic Web Overview

In a *Scientific American* article from May 2001, Tim Berners-Lee et al. introduced their vision of the Semantic Web as an extension to the current World Wide Web (Berners-Lee, et al., 2001). Berners-Lee et al. noted that the current web is a vast collection of individual web page documents that are primarily consumed by human users. Understanding the information that is provided on a web page requires a person's interpretation and reasoning capabilities. The same applies to the process of discovering knowledge and additional document sources. Currently, users have to follow hyperlinks on web pages or submit a keyword query via a search engine in order to find and access resources related to their information need. Until recently, web search algorithms relied solely on the statistical, textual analysis of the document corpus using techniques like synonyms, word stemming and scoring metrics like PageRank to return relevant results based on the entered keywords. The disadvantage of a purely keyword based information retrieval approach is that the meaning of concepts and the relationship between those concepts as expressed in a search phrase or on a web page cannot be extracted and understood by computer agents. For instance, a search engine struggles to give a specific answer to a query like "Which character does Al Pacino play in the movie The Godfather". It lists relevant documents on the first search engine result page that contain the answer, but it is not able to reliably provide one definitive answer as can be seen in Figure 1.1.

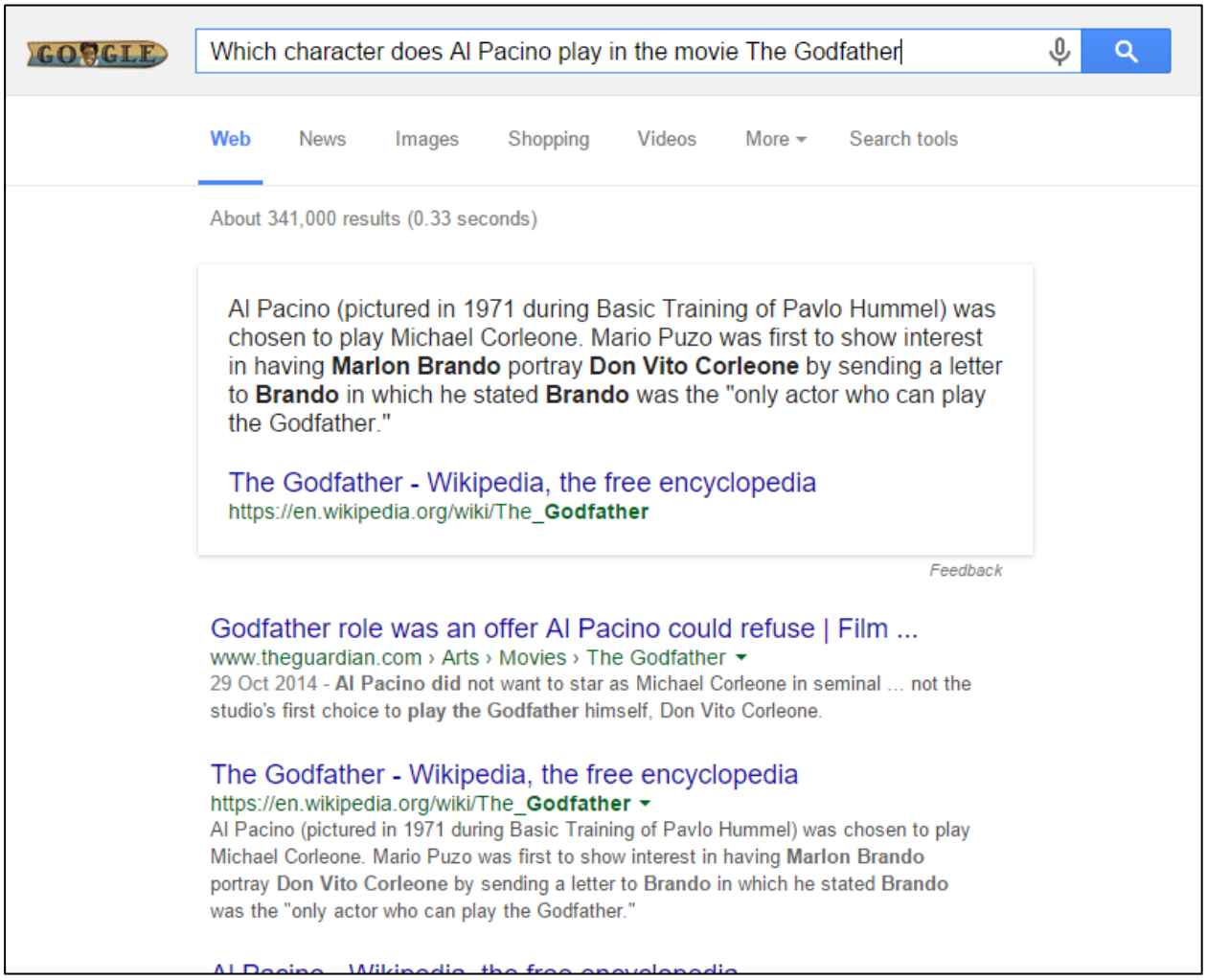

Figure 1.1: Results for the 'semantic' query "Which character does Al Pacino play in the movie The Godfather"

The World Wide Web consortium's (W3C) Semantic Web initiative addresses this shortcoming by introducing a set of technology standards for annotating and linking content on the web. The overall goal is the creation of a 'Web of Data' that can be queried by people as well as machines (W3C, 2015). For this purpose several technologies have been developed: the Resource Description Framework (RDF); a query language for RDF graphs called SPARQL; the web ontology language OWL; and two RDB2RDF mapping languages, namely Direct Mapping and R2RML. The following paragraphs give a brief overview of each of the mentioned technologies as they form the basis for the work undertaken as part of this project.

Knowledge in the Semantic Web is described in the form of RDF triples which are sentence-like constructs, each consisting of a subject, a predicate and an object. RDF triples are used to describe and relate concepts. Multiple RDF triples can form one or more graphs which in turn constitute a RDF dataset (Schreiber & Raimond, 2014). In line with the example above, the following seven RDF triples in TURTLE[1] syntax are one possibility for representing the corresponding information.

```
@prefix foaf:    <http://xmlns.com/foaf/0.1/> .
@prefix dbpedia: <http://dbpedia.org/ontology/> .

_:a
        a                   foaf:Person , dbpedia:FictionalCharacter ;
        dbpedia:movie       <http://www.imdb.com/the_godfather> ;
        dbpedia:performer    :b ;
        foaf:name           "Michael Corleone" .

_:b
        a                 foaf:Person , dbpedia:Actor ;
        dbpedia:movie     <http://www.imdb.com/the_godfather> ;
        foaf:name         "Al Pacino" .
```

The vocabulary used for constructing RDF triples is defined by one or more ontologies. Ontologies define rules about the hierarchy of concepts within a domain of interest and how concepts are related to each other (Hitzler, et al., 2012). For instance, the following two statements in OWL's functional-style syntax define a subclass relationship between the concepts *Person* and *Actor* and the domain and range of the RDF property *perfomer*. The domain of a property describes the concepts that can be used for the subject component of a triple, whilst the range of a property defines the concepts that can be used for the object component.

SubClassOf(dbpedia:Actor foaf:Person)
ObjectPropertyDomain(dbpedia:performer dbpedia:FictionalCharacter)
ObjectPropertyRange(dbpedia:performer dbpedia:Actor)

The use of ontologies allows for automated reasoning via inference rules which makes them a powerful knowledge modelling tool as a dataset does not need to contain explicit knowledge. Again, this can be demonstrated with the above example. If the triple `_:b a dbpedia:Actor` were not present in the dataset, the class assertion could still be made based on the above property axioms.

SPARQL is a language for querying, manipulating and constructing RDF data. Its query syntax is similar to SQL and it supports a number of syntactically identical query clauses like GROUP BY, HAVING, DISTINCT and ORDER BY. A typical SPARQL query evaluates a set of triple patterns and identifies matching RDF triples in the dataset. The values of matching triples that are bound to all or a subset of variables used in the triple patterns are projected as results (Harris & Seaborne, 2013). To keep with

[1] http://www.w3.org/TR/turtle/

the example above, the following SPARQL query issued against the RDF dataset provides an answer to the original question.

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix dbpedia: <http://dbpedia.org/ontology/>

SELECT ?y
WHERE { ?x  foaf:name           ?y .
        ?x  dbpedia:performer   ?z .
        ?x  dbpedia:movie       <http://www.imdb.com/the godfather> .
        ?z  foaf:name           "Al Pacino" .
        ?z  a                   dbpedia:Actor . }
```

The result returned for the above query is shown in Table 1.1 below. It shows that there is one result in which the projection variable "y" is bound to the literal value "Michael Corleone".

y
Michael Corleone

Table 1.1: SPARQL query result

Finally, two RDB2RDF mapping languages define how contents of a relational database can be converted into a virtual or materialised RDF representation. In the case of Direct Mapping, RDF triples are generated based on the entities, their attributes and relationships as well as the data in the input database (Arenas, et al., 2012). In contrast, when using R2RML, the vocabulary of existing ontologies is incorporated in a mapping document specifying the exact class and property names that are used for transforming relational tuples to RDF triples (Das, et al., 2012). As a result, RDF datasets produced in the latter way are easier to query in case well known vocabularies are utilised. Additionally, such datasets can be readily integrated with other knowledge sources and published on the web, thereby becoming part of what Semantic Web researchers refer to as 'Linked Data'.

1.2 Problem Statement and Solution Approaches

The commoditization of information technology both in the business and consumer domains led to the creation and storage of vast amounts of data which is likely to continue to grow in size and volume in the future. This data exists in various formats and is generally grouped into two high-level categories; structured and unstructured data.

Structured data can be described as information that is maintained in a specific format and that has meta-data associated with it. Examples for structured data are data contained in spreadsheets, databases or other organised file structures. In contrast, text and media files, like images, audio and videos are considered unstructured data.

The W3C's vision for the Semantic Web includes the publication of knowledge and information in a machine-readable way on the WWW (Berners-Lee, et al., 2001). For this to happen, the W3C acknowledged that data needs to be accessible online, to be published in a standard format and to be interlinked through RDF uniform resource identifiers (URIs). This approach is summarised under the term 'Linked Data' which was introduced in the previous section. The technology used for this purpose is RDF which was also introduced in the previous section.

Heath and Bizer (2011) present six approaches for publishing Linked Data in the form of RDF datasets and how such datasets can be created from structured and unstructured data sources which are summarised in Figure 1.2 (Heath & Bizer, 2011).

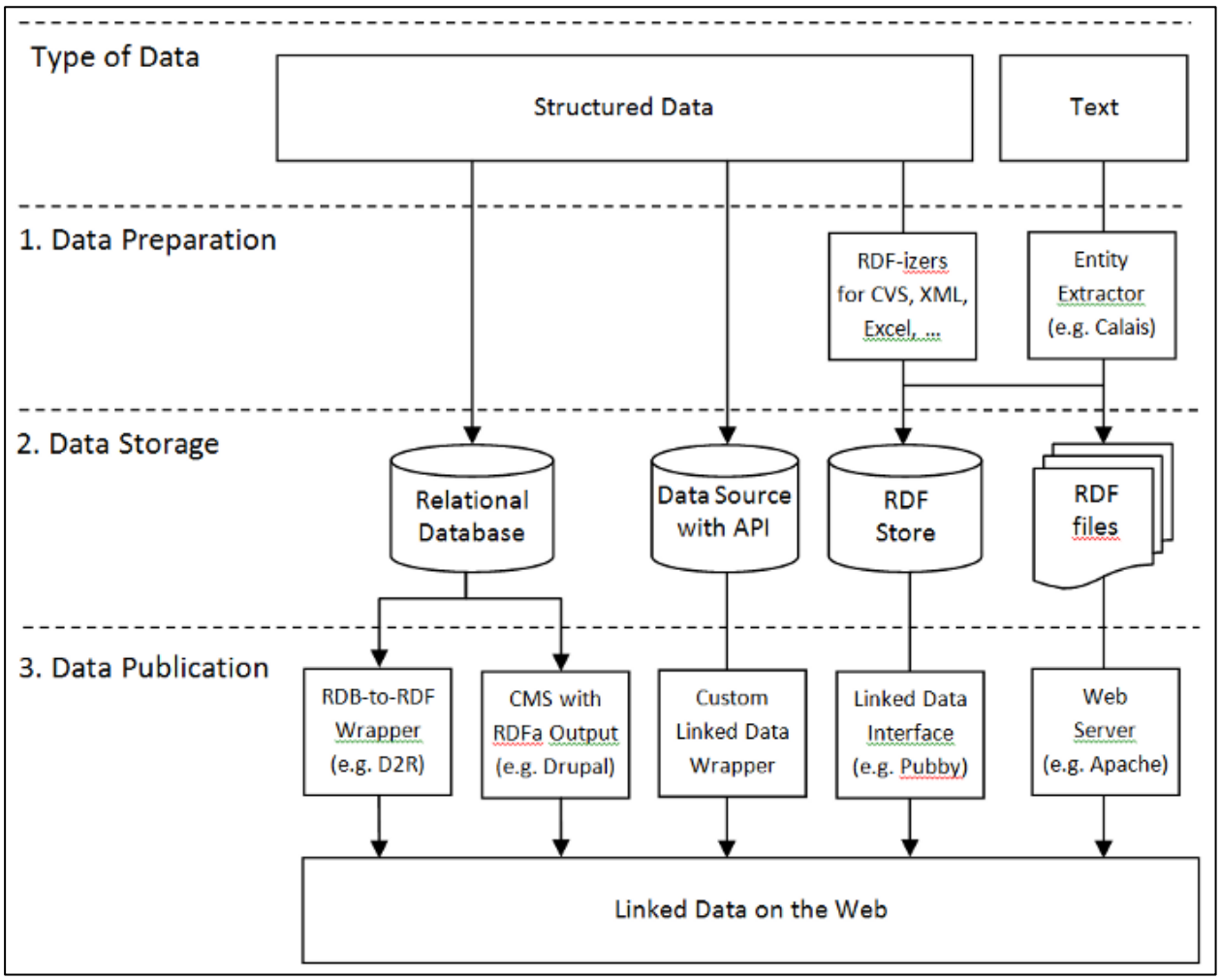

Figure 1.2: Linked Data Publishing Options (Heath & Bizer, 2011)

The most straightforward method for publishing Linked Data on the web is hosting RDF files on a web server. This approach requires the data to either be in RDF representation natively, or for it to be transformed into RDF by an appropriate tool or process. In case the source data is static, this option is very well suited for the publication of information as Linked Data.

An alternative to storing RDF datasets as plain text files are RDF stores, also known as triplestores. They are usually implemented based on a relational database management system (RDBMS) with the objective to take advantage of the latter's efficient query algorithms and reliable transaction management (Chebotko, et al., 2009). A potential downside of this approach is the complex translation of SPARQL queries into SQL queries on the relational triplestore which is required for providing a Linked Data interface to the RDF store.

Heath and Bizer (2011) also list the custom development of two kinds of wrappers for publishing existing content as Linked Data. The first wrapper solution refers to a scenario in which web pages are generated dynamically on the server, and aims at not only creating HTML code server-side, but also RDF output which is deployed to the web server simultaneously. A similar approach is proposed for data that is made available via Application Programming Interfaces (APIs), in order to serve requests for RDF serialised data as well, instead of restricting the response to the API's native format.

Another option is annotating content in HTML pages with RDFa. That way RDF triples can be extracted directly from web pages with the help of appropriate parsing tools and made available as Linked Data.

Lastly, RDF datasets can be generated from contents in a relational database by using mapping templates to translate table records into triples.

Relational databases are widely used as the backend for content storage, not just for web sites, but also for other applications due to the technology's maturity and reliability. On closer inspection of the above data sources, it can be argued that most of them are most likely relational databases. For instance, many web content management systems rely on a RDBMS for content storage. Content that is made accessible via a Web API is usually also held in a database. Data used for creating RDF datasets that are then imported into a triplestore, or exported to flat RDF files, was extracted from a relational database at some stage as well. This is due to the fact that commonly spreadsheets or XML files contain data which was exported from a relational database via a reporting function.

In order to achieve the objective of publishing existing knowledge and information as Linked Data, it is evident that a method for translating relational data into an RDF serialisation would make a significant contribution to this goal. Addressing this requirement, several Semantic Web researchers developed their own processes and mapping languages (Bizer, 2003; Perez de Laborda & Conrad, 2006; Lv, et al., 2010; Sequeda & Miranker, 2013). Additionally, the W3C's RDB2RDF working group released two specifications called Direct Mapping and R2RML that have since been adopted as the de-facto standards for generating RDF datasets from relational data (Das, et al., 2012).

With the standardisation of mapping languages, implementers are left with the decision of how to provide access to the RDF serialisation of relational data.

One option is materialising RDF triples and either storing them to one or more files or loading them into a triplestore. The former alternative has the advantage that static RDF files can be queried directly with SPARQL, while such queries need to be translated to SQL queries against the triplestore schema in the latter case which proved to be a complex undertaking (Chebotko, et al., 2009). Both approaches suffer from the issue of data duplication and the associated additional storage requirements as well as a processing overhead for synchronising updates in the database to the materialised RDF dataset. In case the relational dataset in question is small and static or changes very infrequently, these two choices are a good fit nevertheless.

In most scenarios, however, providing a virtual view on the relational data's RDF representation, which can be queried via SPARQL without materialising triples, is a better solution. One advantage is that only one data source needs to be maintained and consequently this master instance serves as the single source of truth. In contrast to periodical updates, achieved by taking snapshots of the data, transforming it and synchronising the RDF representation, the virtual approach guarantees that both views, relational and RDF, are consistent and up-to-date. Non-materialisation also provides the flexibility of altering the underlying mapping specification without the need to re-create the entire RDF dataset. For instance, if the database schema changes, this will only need to be reflected in the mapping file. The disadvantage of the virtual approach is the additional overhead introduced by the requirement to translate SPARQL queries into SQL queries against the original database schema under consideration of the mapping specification. Empiric research indicates, though, that the larger the dataset is, the lower the translation overhead is compared to the native SPARQL query processing overhead (Bizer & Schultz, 2011).

In summary, it can be argued that publishing information stored in a relational database as Linked Data is a critical issue which has been partly addressed by the two mapping standards that were developed and

recommended by the W3C. However, the question of how to provide efficient access to the RDF representation of data in a RDBMS remains. In the past, many research activities focused on the efficient storage and retrieval of triples in an RDF store (Chong, et al., 2005; Chebotko, et al., 2009; Elliot, et al., 2009). With the possibility of providing virtual views on RDF datasets via the mentioned mapping specifications, the focus shifted towards algorithms for translating SPARQL queries into SQL queries based on a Direct Mapping or R2RML mapping (Unbehauen, et al., 2012; Priyatna, et al., 2014).

1.3 Overview and Example of Implemented Solution

The solution developed as part of this dissertation project enables SPARQL query execution over a relational database by translating the SPARQL query under consideration of R2RML mapping definitions into an equivalent SQL query against the database schema. During this process no RDF triples are materialised, neither in a physical or virtual manner.

This section was deliberately kept brief and therefore refers to notation and terms which are explained in more detail in section 2.1.

At a high-level, the process comprises the following steps. First, the application reads a Java properties file containing configuration parameters like database connection details and file paths to the R2RML mapping document and the SPARQL query file. In the next step, the contents of the R2RML mapping file and the SPARQL query file are parsed and stored as Java objects in memory. The query translation algorithm then analyses the triple patterns (triple statements containing variable placeholders) and projection variables (all or a subset of the variables used in triple patterns) in the SPARQL query, and identifies the database entities, attributes and selection conditions, which need to be incorporated into a SQL query, by processing matching term maps (rules defining how triples are created from relational records) in the R2RML mapping file. The assembled SQL query is executed against the database via JDBC and the returned result set is output to the console prompt.

The process detailed above can be demonstrated with the following example in which the relational database contains only one table named “movies” which is defined as described in Table 1.2.

Column name	Data type
id	INTEGER
title	TEXT
production_year	INTEGER

Table 1.2: DB Table Example

The R2RML mapping file contains the definition of one triple map which consists of one subject map and two predicate object maps.

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix mo: <http://www.movieontology.org/2009/10/01/movieontology.owl#> .
@prefix mo2: <http://www.movieontology.org/2009/11/09/movieontology.owl#> .
@prefix dbpedia: <http://dbpedia.org/ontology/> .

<#MovieTriplesMap>
    rr:logicalTable [ rr:tableName "title" ];

    rr:subjectMap [
      rr:template "http://www.imdb.com/movie_{id}";
      rr:class mo2:Movie;
      rr:class dbpedia:Film;
      ];
```

```
    rr:predicateObjectMap [
      rr:predicate            mo:title ;
      rr:objectMap            [ rr:column "title" ];
    ];

    rr:predicateObjectMap [
      rr:predicate            mo:releasedate ;
      rr:objectMap            [ rr:column "production_year" ];
    ];
```

If the mapping definition were to be used for materialising triples, each record in the database table would result in four triples of the following form.

```
<http://www.imdb.com/movie_{value from column id}>
      a                 mo2:Movie ;
      a                 dbpedia:Film ;
      mo:title          {value from column title} ;
      mo:releasedate    {value from column production_year} .
```

Considering the RDF triple template above and the following SPARQL query, it can be seen that the answer to the latter are the values in columns "title" and "production_year" projected for all records in table "movies".

```
prefix mo: <http://www.movieontology.org/2009/10/01/movieontology.owl#>
prefix mo2: <http://www.movieontology.org/2009/11/09/movieontology.owl#>

SELECT ?y ?z
WHERE { ?x  a                   mo2:Movie .
        ?x  mo:title            ?y .
        ?x  mo:releasedate      ?z . }
```

Based on the above inputs, the query translator program would translate the SPARQL query into a corresponding SQL query and return the query results[2] as detailed on the screenshot in Figure 1.3.

```
SPARQL to SQL Query Translator. Run with -h for help on options.

(SELECT * FROM movies) AS MovieTriplesMap ==> added by http://www.w3.org/1999/02/22-rdf-syntax-ns#type
(SELECT * FROM movies) AS MovieTriplesMap ==> added by http://www.movieontology.org/2009/10/01/movieontology.owl#title
MovieTriplesMap.title AS y ==> added by http://www.movieontology.org/2009/10/01/movieontology.owl#title
(SELECT * FROM movies) AS MovieTriplesMap ==> added by http://www.movieontology.org/2009/10/01/movieontology.owl#releasedate
MovieTriplesMap.production_year AS z ==> added by http://www.movieontology.org/2009/10/01/movieontology.owl#releasedate

Running following query against database.

SELECT MovieTriplesMap.title AS y,
       MovieTriplesMap.production_year AS z
FROM   (SELECT * FROM movies) AS MovieTriplesMap

y                                           z
12 Angry Men                                1957
Fight Club                                  1999
Il buono, il brutto, il cattivo.            1966
Pulp Fiction                                1994
Schindler's List                            1993
The Dark Knight                             2008
The Godfather                               1972
The Godfather: Part II                      1974
The Lord of the Rings: The Return of the King     2003
The Shawshank Redemption                    1994

Total results found: 10

Done.
SPARQL to SQL Query Translator. Processing completed. Check log file.
```

Figure 1.3: Query Translator output

[2] The movies view in this example contains only ten records rather than the whole IMDB database.

The remainder of this report will provide a definition of key concepts, R2RML mapping details, a description of the application's technical specification, and the technologies utilised (Section 2), present the implementation of the translation algorithm (Section 3), provide validation in form of test cases and results (Section 4), undertake a critical evaluation of the outcomes by highlighting known issues and possible future improvements (Section 5) and close with the presentation of a conclusion (Section 6).

2 Specification and Design

2.1 Definitions and Scope

Before providing detail on the relational schema on whose basis the R2RML mapping document was created, as well as the technical specification of the developed application, it is necessary to introduce and define some of the key concepts and syntax used in the remainder of this report.

2.1.1 RDF Terms and Triples

There are three types of RDF terms (T), namely International Resource Identifiers or IRIs (I), literals (L) and blank nodes (B). Unbehauen et al. (2012) express this relationship formally as

$$T = I \cup L \cup B$$

A RDF triple is a sentence like structure consisting of one subject, one predicate and one object, whereby the subject is either an IRI or a blank node, the predicate is an IRI and the object is an IRI, literal or blank node. Using the above notation, a triple can be denoted as

$$(tr_s, tr_p, tr_o) \in (I \cup B) \times I \times (I \cup L \cup B)$$

For the purpose of this project, the scope of RDF terms is limited to IRIs and literals. Blank nodes are excluded as mapping definitions were restricted to generating triples consisting of IRIs and literals only. Hence, all RDF triples considered by the query translation algorithm are elements of the following Cartesian product

$$(tr_s, tr_p, tr_o)^* \in I \times I \times (I \cup L)$$

2.1.2 R2RML Triple and Term Maps

The R2RML mapping document is made up of several so called *triple maps*. Each triple map describes how one row from a database table, view or the result set of a SQL query is transformed into one or more RDF triples. This is achieved through the combination of different *term maps* into one triple map (Das, et al., 2012).

Term maps are differentiated into subject maps, predicate maps and object maps which define how the corresponding RDF term for the subject, predicate and object position of a triple is generated. A triple map usually contains one subject map and pairs of predicate and object maps grouped into one or more predicate object maps. A basic example of a complete triple map can be found in section 1.3.

R2RML offers three alternatives for generating the RDF term for a triple component. The first option is specifying a constant literal or IRI in the mapping document. A so called constant-valued term map ensures that every triple generated for one relational record has an identical value for the triple component controlled by this term map. Alternatively, a column-valued term map assigns the value of a tuple element retrieved from the relational record to the triple component. In most cases this value will be a literal, but theoretically it could be an IRI as well. Thirdly, a template-valued term map provides a method for combining one or more column values from a relational record and merging them with a string template. Such term maps are mostly used to create IRIs with database primary key values which uniquely identify a resource, or lookup values which are mapped to resources, or IRIs, from an existing ontology vocabulary.

For the purpose of this project, the R2RML mapping document will restrict term map usage to the following combinations in order to maintain the scope defined in section 2.1.1.

All subject maps are using template-valued term maps for generating an IRI from a string template specific to the resource created and a key value uniquely identifying the resource. This means all subjects generated via the R2RML mapping definition are valid IRIs. An additional restriction is that the template map references only one column.

All predicate maps are using constant-valued term maps reusing an existing object or data property from an applicable ontology vocabulary for generating the predicate component of a triple.

Object maps are using either column-valued or template-valued term maps. In the former case the object component will be a literal derived from a database record's column value. The latter case generates an IRI that is identical to the IRI generated by a subject map of a different triple map and thereby translates a foreign key relationship into a triple relating the two resources with each other via a property. This implies that the template map references only one table column as in the subject's case.

Finally, all triples belong to the default graph of the RDF dataset generated. No named graphs are produced.

The R2RML classes GraphMap, Join, RefObjectMap and properties child, graph, graphMap, inverseExpression, joinCondition, parent and parentTriplesMap are defined as out of scope to reduce the complexity of the query translation algorithm.

2.1.3 SPARQL Triple and Graph Patterns

SPARQL queries evaluate one or more *triple patterns* for finding matching triples in the queried RDF dataset. Triple patterns are triples that contain variable placeholders (V) instead of an RDF term for some or all of the triple components. They are formally denoted as

$$(tp_s, tp_p, tp_o) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup L \cup B \cup V)$$

Considering the limitation of scope defined in section 2.1.1 and adding two further restrictions in not using variables for the predicate component, and no IRIs for the subject component of triple patterns their elements are defined as

$$(tp_s, tp_p, tp_o)^* \in V \times I \times (I \cup L \cup V)$$

The rationale behind the scope limitation with respect to the predicate component is that all predicates are IRIs of object or data properties as per definition in section 2.1.2. No database values are taken into account when generating these IRIs. Allowing predicate variables would increase the set of possible solutions significantly and add additional processing overhead as well as complexity. As all IRIs are generated from a template map and primary key column, reducing the subject component scope to variables is a minor limitation. Users would usually not know the exact IRI in this scenario, and are more likely to search for an attribute that identifies the resource.

A typical SPARQL query contains several triple patterns that form a *basic graph pattern* (Harris & Seaborne, 2013). The solutions to a SPARQL query are all triples from the RDF dataset forming a graph that matches the graph pattern expressed in the query. The values bound to the variables in each solution are processed by SPARQL modifiers to be projected, removed or returned in a certain order. Furthermore, it is possible to define a portion of the graph pattern as optional, thereby allowing

solutions that only match the graph pattern partially. Filter clauses refine solutions to a subset meeting specified constraints. These constraints can be Boolean expressions or checks for existence or non-existence of certain triples.

Table 2.1 (Harris & Seaborne, 2013) summarises SPARQL's abstract syntax. The terms highlighted in bold are in scope for the query translation algorithm. The main focus of this project is on translating projection queries over different graph patterns. While the translation of the SPARQL modifiers detailed below to equivalent SQL syntax is straightforward, the transformation of graph patterns into SQL selection statements and corresponding conditions is the main challenge. These issues will be addressed in section 3.

Patterns	Modifiers	Query Forms	Other
RDF terms	**DISTINCT**	**SELECT**	VALUES
Property path expression	REDUCED	CONSTRUCT	SERVICE
Property path patterns	**Projection**	DESCRIBE	
Groups	**ORDER BY**	ASK	
OPTIONAL	LIMIT		
UNION	OFFSET		
GRAPH	Select expressions		
BIND			
GROUP BY			
HAVING			
MINUS			
FILTER			

Table 2.1: SPARQL abstract syntax tree (Harris & Seaborne, 2013)

2.2 R2RML Mapping Document

Besides a SPARQL query expressed in the syntax that was introduced in the previous section, the query translator application has a second input in form of a R2RML mapping document. The mapping was created for a number of custom views on the IMDB database whose entity relationship diagram is detailed in Figure 2.1 below.

The exact SQL queries that were used to create the views are referenced in Appendix B. The views are based on joins between several base tables in order to de-normalise some of the data that was stored in lookup tables, and to cast other values into appropriate data types.

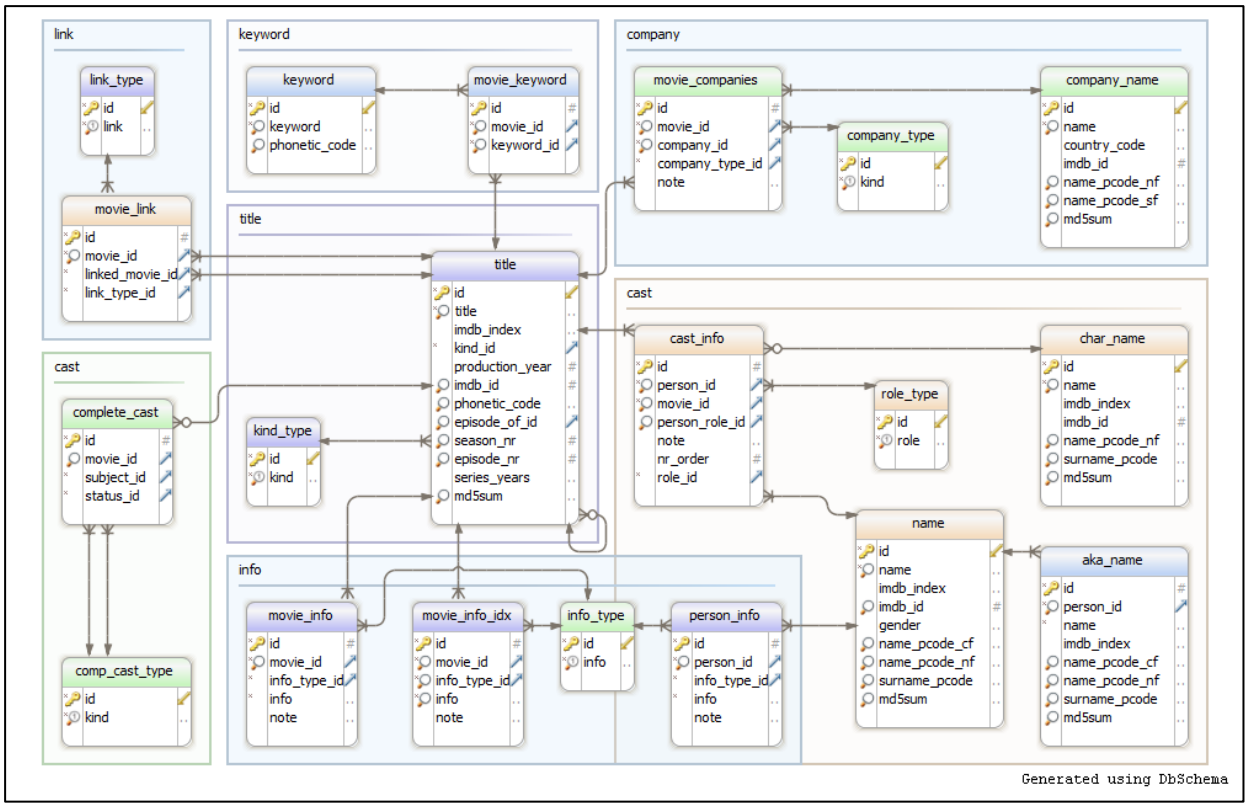

Figure 2.1: IMDB Entity Relationship Diagram

Vocabularies from several ontologies, including dbpedia[3], schema.org[4], Friend of a Friend (foaf[5]) and the movie ontology[6] were used to define the mappings. An overview of the classes and properties that were incorporated into the mapping document are shown in graph representation in Figure 2.2.

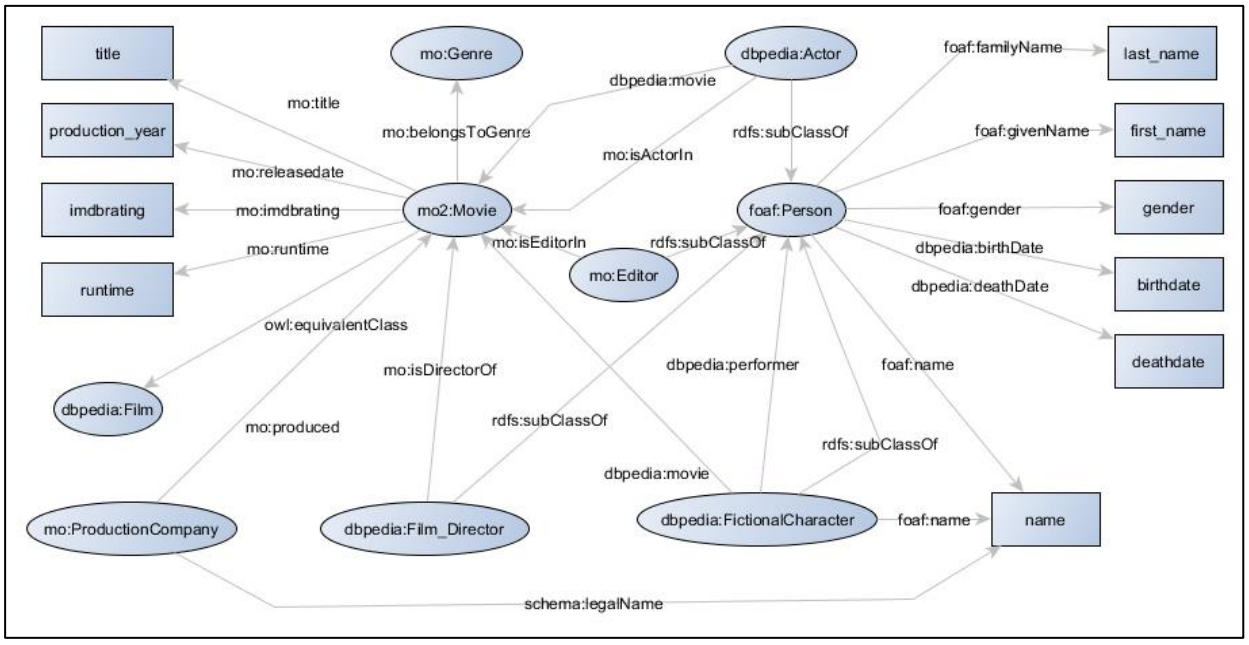

Figure 2.2: Graph representation of mapped ontology classes and database columns

[3] http://wiki.dbpedia.org/services-resources/ontology
[4] https://schema.org/
[5] http://xmlns.com/foaf/spec/
[6] http://www.movieontology.org/

The complete mapping document can be found in Appendix C. Table 2.2 gives a summary of individual base table (or view) columns, which term map they were mapped to and, if applicable, the predicate referenced in the predicate object map.

View	Column	Term map	Predicate	Term Map Type
movies	id	subjectMap	n/a	template-valued
	title	objectMap	mo:title	column-valued
	production_year	objectMap	mo:releasedate	column-valued
	imdbrating	objectMap	mo:imdbrating	column-valued
	runtime	objectMap	mo:runtime	column-valued
moviegenres	id	subjectMap	n/a	template-valued
	genre	objectMap	mo:belongsToGenre	template-valued
persons	id	subjectMap	n/a	template-valued
	name	objectMap	foaf:name	column-valued
	last_name	objectMap	foaf:familyName	column-valued
	first_name	objectMap	foaf:givenName	column-valued
	gender	objectMap	foaf:gender	column-valued
	birthdate	objectMap	dbpedia:birthdate	column-valued
	deathdate	objectMap	dbpedia:deathdate	column-valued
actors	person_id	subjectMap	n/a	template-valued
	movie_id	objectMap	dbpedia:movie mo:isActorIn	template-valued
directors	person_id	subjectMap	n/a	template-valued
	movie_id	objectMap	mo:isDirectorOf	template-valued
editors	person_id	subjectMap	n/a	template-valued
	movie_id	objectMap	mo:isEditorIn	template-valued
companies	id	subjectMap	n/a	template-valued
	name	objectMap	schema:legalName	column-valued
	movie_id	objectMap	mo:produced	template-valued
characters	person_role_id	subjectMap	n/a	template-valued
	name	objectMap	foaf:name	column-valued
	movie_id	objectMap	dbpedia:movie	template-valued
	person_id	objectMap	dbpedia:performer	template-valued

Table 2.2: Mapping summary

For the query translator application to work, the triple map names in the R2RML mapping document need to start with the "#" character. This is necessary for the application to extract the triple map name from the base IRI configured in the Java properties file.

Furthermore, it is assumed that all template maps are only referencing one column field and are not using two or more columns for generating an IRI.

2.3 Technical Specification

The application was developed as a Java console application and consists of three separate components:

1. R2RML Parser v0.7-alpha[7]
2. Sesame Query Parser API 2.8.4[8]
3. Query translation module

[7] https://github.com/nkons/r2rml-parser
[8] http://rdf4j.org/

R2RML Parser by Nikolaos Konstantinou was chosen for the R2RML parsing component because it is an open-source application written in Java which could easily be extended with the Sesame API for SPARQL query parsing and the query translation module developed as part of this project. It is explicitly highlighted at this point that R2RML Parser's application structure and source code has been reused as the basis for the query translation program. In particular, the class implementations in Table 2.3 that were developed as part of R2RML Parser were adopted for this project.

R2RML Parser class implementation	File name	Detail on reuse
Main	Main.java	The Main class implementation is reused as the Main class for the Query Translator application. However, lines 99 to 104 (referring to the original, unmodified Main class) have been removed to prevent the generation of an RDF output file. Additional code has been added to integrate the query translation module.
Database	Database.java	The Database interface implementation is reused for processing the SQL query which was derived from the translation of the input SPARQL query.
Parser	Parser.java	The Parser class implementation is reused for parsing the R2RML mapping document and passing its parsed representation to a MappingDocument object.
MappingDocument	MappingDocument.java	The MappingDocument class is reused for storing a parsed representation of the R2RML mapping file.

Table 2.3: Reused R2RML classes

As detailed in Figure 2.3 there are three inputs that have to be provided to the query translator application. A Java properties file stores various configuration parameters like the database connection string, the log file name and the syntax of the mapping file as well as the paths to the SPARQL query file and the R2RML mapping document.

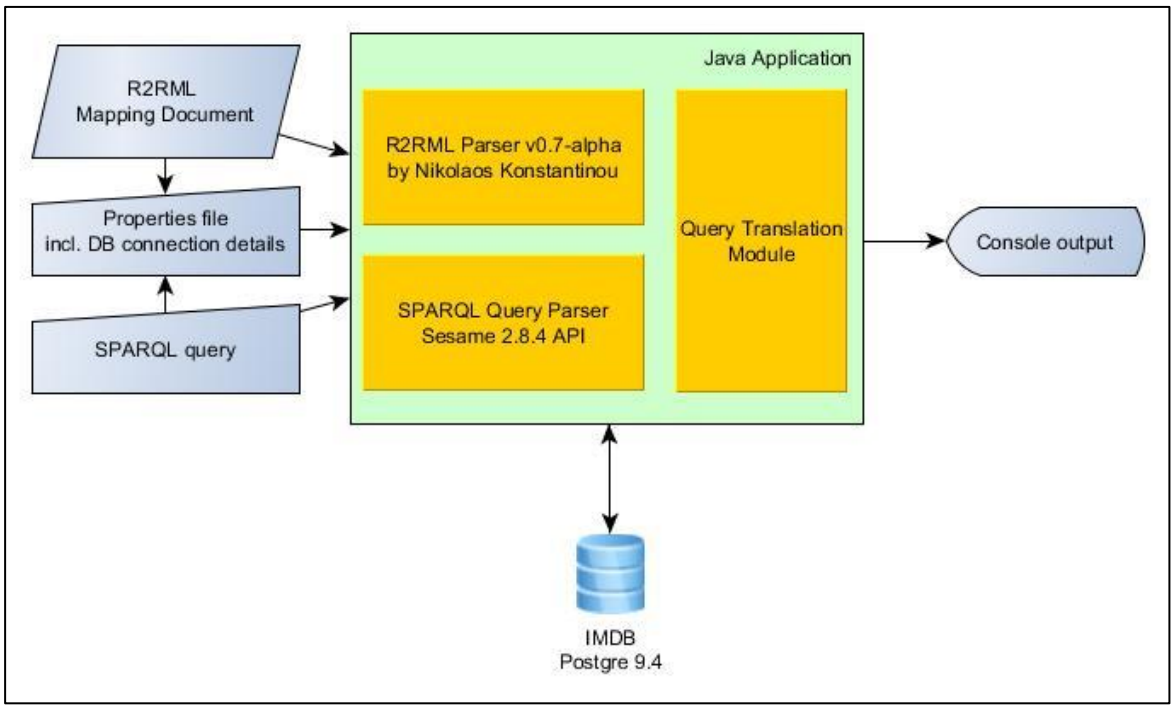

Figure 2.3: Query Translator Architecture

The application retrieves these values from the properties file, continues with reading the mapping document from disk and storing its parsed representation in a MappingDocument object. In the next step the SPARQL query file is read from disk, parsed and stored in a SPARQLParser object. The two objects are then passed to the query translation module responsible for the transformation of the SPARQL query to a SQL query. During processing key steps are written to the console interface and log file. Upon completion, the query translation module returns the translated query to the main program which executes it against the database. The results returned by the SQL query are output to the console and log file. For readability, the results will be returned in a tabular format rather than one of the response formats specified for SPARQL like XML or JSON.

The details on the translation algorithm are provided in section 3 of this report.

2.4 Utilised Technologies

During the completion of this project several software packages were utilised to assist with various tasks. Eclipse IDE was used for Java development and, in conjunction with the Maven plugin, for the purpose of importing and managing library dependencies and building the program executables.

The IMDB database[9] was imported into a PostgreSQL database backend with the help of a Python application called IMDbPy[10]. Toad for Eclipse plugin was used to run queries and inspect schema objects from within Eclipse.

The diagrams in this report were created with yEd Graph Editor[11], except for the entity-relationship diagram which was created with DbSchema[12].

[9] http://www.imdb.com/interfaces
[10] http://imdbpy.sourceforge.net/
[11] http://www.yworks.com/en/products/yfiles/yed/
[12] http://www.dbschema.com/

3 Implementation of the Query Translation Algorithm

3.1 High-level Overview

The translation process has to establish the composition of the SQL query's SELECT and FROM clauses at a minimum and, depending on the triple patterns in the SPARQL query, potentially also needs to determine the conditions in the WHERE clause. Additionally, it might need to add the DISTINCT keyword and an ORDER BY clause. In order to achieve these goals, the algorithm is required to find the triple maps in the R2RML mapping document that contribute to the solution, and reverse-engineer the names of base tables and columns in the relational database schema that provide specific values for the solution. Once all the components are established, the algorithm assembles the complete query and passes it back to the main program for execution. These high-level steps can be broken down into the following individual tasks.

1. Identify triple maps based on individual triple patterns in SPARQL query
2. Add base table, view or SQL query from identified triple maps to FROM clause with the triple map name as the table identifier
3. Identify the variables in triple patterns that are projected by the query, and find the base table columns that are used for generating the corresponding triple component
4. Add column names with identifiers named after projection variable to SELECT clause
5. Establish whether a WHERE clause is required and, if that is the case, determine and add conditions to it
6. If the SPARQL query contains an OPTIONAL graph pattern, determine conditions for WHERE clause or add outer joins to FROM clause depending on graph pattern
7. If the SPARQL query contains a FILTER clause, determine its type (EXISTS, NOT EXISTS, Boolean constraint comparing variable and literal, Boolean constraint comparing two variables) and add corresponding conditions to WHERE clause
8. If the SPARQL query contains a DISTINCT query modifier, add it to SELECT clause
9. If the SPARQL query contains an ORDER BY clause, add it after the WHERE clause

A reference to the Java source code files implementing the above steps is included in this report in Appendix E. The following section provides a more detailed explanation of individual steps.

3.2 Algorithm Details

The query translation algorithm is implemented as a Java class containing the properties and methods detailed in Table 3.1.

QueryTranslator	Explanation
allVarTrack	List of all variable names that are present in SPARQL query triple patterns
fromClause	String variable to store assembled FROM clause of translated SQL query
fromClauseSet	Set of table names or queries and their identifiers that will be added to FROM clause
log	Object to control logging
objectValue	String variable to store IRI, literal value or variable name of object in triple pattern
predicateIRI	String variable to store predicate IRI value of triple pattern
projVarQueue	List for queueing projection variables for post

QueryTranslator	Explanation
	processing
projVarTrack	List tracking the associated triple map and base table column name for each projection variable
selectClause	String variable to store assembled SELECT clause of translated SQL query
selectClauseSet	Set of base table columns and their identifiers that will be added to SELECT clause
sqlQuery	String variable to store completely translated SQL query
subjectVar	String variable to store subject variable name in triple pattern
subVarQueue	List for queueing subject and object variables for post processing
subVarTrack	List for tracking the subject and object variable, associated triple map and base table or view for each triple pattern in SPARQL query
tableView	String variable to store base table, view or SQL query extracted from triple map
tripleMapStr	String variable to store triple map's name which is used as an identifier in the FROM clause and for column references in SELECT and WHERE clauses
whereClause	String variable to store assembled WHERE clause of translated SQL query
whereClauseSet	Set of conditions that will be added to WHERE clause
evalFilter	Method for evaluating Boolean comparison and regular expressions in FILTER clauses and translating them into corresponding selection conditions based on relational entities; returns a list of conditions that are added to the WHERE clause
getColumn	Method for retrieving the column name that has been identified for a triple pattern variable from the tracking list
getColumnName	Method for retrieving the column name by extracting it from a predicate object map based on an input of a triple map and a predicate
getTMforVar	Method for retrieving the triple map that has been identified for a triple pattern variable from the tracking list
getTMSubjectTemplate	Method for retrieving the subject template for a specific triple map
matchingTM	Method for returning all triple maps that contain the stated predicate map
noOfMatchesTM	Method for retrieving the number of triple maps that contain the stated predicate map
translate	The translation method which is called by the main program

Table 3.1: QueryTranslator Java class

The query translation algorithm has nine main stages which are executed in order to process the query modifiers and patterns provided in the SPARQL input query. The nine stages can be described as follows.

1. Variable collection
2. Projection processing
3. Post processing for FROM clause
4. Post processing for SELECT clause
5. EXISTS and NOT EXISTS FILTER processing
6. SELECT clause assembly
7. FROM clause assembly
8. WHERE clause assembly
9. Finalising SQL query

The translation of OPTIONAL patterns originally included in the scope has been omitted due to the approaching submission deadline.

3.2.1 Variable Collection

During the variable collection stage, a list with the name of all projection variables and a list with all variable names occurring in triple patterns in the SPARQL query are created. The former list is used to add the projection variables in the right order to the SELECT clause in the SQL query, while the latter list is traversed several times in subsequent stages to find out which position a variable is used in a triple pattern.

Pseudo code:

listProjectionVariables ← projection variable names in SPARQL query
listAllQueryVariables ← all variable names used in SPARQL query

3.2.2 Projection Processing

The projection processing stage iterates over every triple pattern in the SPARQL query and identifies the triple map in the R2RML mapping document that would generate the triple expressed through the pattern. This is achieved by iterating over every triple map in the mapping document and determining the triple map name by comparing the predicate IRI from the triple pattern in the query with the predicate maps in each triple map. As defined in section 2.1.3, the scope for predicates in a triple pattern is limited to IRIs, hence reducing the complexity of the algorithm for finding the correct triple map. There are two scenarios that need to be considered at this stage.

The first possibility is that the predicate IRI is *rdf:type*. In that case, it is also necessary to scan the subject map and compare the class IRI to the IRI stated as the object in the triple pattern. If they match, the triple map's logical table and its name are added to the FROM clause. Additionally, if the subject variable is in the list of projection variables, the base table column name extracted from the subject map's template map is added to the SELECT clause. Since *rdf:type* could also be produced by a predicate object map, the same process is applied for these term maps. It is highlighted that the translation algorithm cannot process triple patterns where the *rdf:type* predicate is used with a variable in the object position as this scenario could not be implemented in time.

The second scenario is that the predicate IRI is not *rdf:type*. As a result, predicate object maps are scanned to find the predicate map containing the corresponding IRI from the triple pattern. As part of the search, it is also necessary to establish the count of candidate triple maps for the predicate as it is possible that a predicate IRI is used in predicate maps of different triple maps. If a triple map can be uniquely identified (predicate IRI occurs only in one predicate map in whole mapping document), the

triple map's logical table and its name are added to the FROM clause. As above, if the subject variable is a projection variable, the column name in the template map is added to the SELECT clause. However, if there is more than one matching triple map, it is first checked whether the subject variable has been processed in a previous iteration, in which case a second evaluation is not necessary. If that is not the case, the subject variable in the triple pattern needs to be queued for the post processing stage.

The previous steps might have processed some of the projection variables already, but it is necessary to identify all of them. Therefore, the list of projection variables is traversed. First, it is checked whether the projection variable has been processed already. If this is not the case, it is established whether the variable is used in the subject position of a triple pattern. If so, the base table column name is extracted from the subject map's template map and added to the SELECT clause. The other possibility is for the variable to be in the object position. Again, it needs to be checked whether the predicate is used in more than one predicate map in the mapping document, as it is otherwise not possible to uniquely identify the triple map. If the predicate IRI is found in only one predicate map, its column map or template map are used to identify the base table column name which is added to the SELECT clause. If there is more than one candidate triple map, the projection variable needs to be added to a queue for post processing.

Pseudo code:

```
for each triple pattern in SPARQL query
  sv ← subject variable name
  pn ← predicate IRI
  ov ← object variable name, IRI, or literal

  for each triple map in mapping document
    tm ← triple map name
    bt ← base table name or sql query
    if pn is rdf:type then
      if ov is identical to one of the class IRIs in subject map of tm then
        if sv in listProjectionVariables then
          SELECT clause ← getColumnFromTemplate (sv)
          trackingListProjectionVariables ← sv, getColumnFromTemplate (sv)
        FROM clause ← bt as tm
        variableTrackingList ← add(sv,bt,tm)
    else
      for each PredicateObjectMap in tm
        if pn is not rdf:type then
          if countNoOfTripleMaps(pn) = 1 then
            if sv in listProjectionVariables then
              SELECT clause ← getColumnFromTemplate(sv)
              trackingListProjectionVariables ← sv, getColumnFromTemplate (sv)
            FROM clause ← bt as tm
            variableTrackingList ← add(sv,bt,tm)
          else if countNoOfTripleMaps(pn) > 1 then
            if sv not in trackingListProjectionVariables then
              queueListSubjectVariables ← sv
          for each variable in listProjectionVariables
            if variable not in trackingListProjectionVariables then
              if variable = sv then
                SELECT clause ← getColumnFromTemplate(sv)
                trackingListProjectionVariables ← sv, getColumnFromTemplate (sv)
              else if variable = ov then
```

```
            if countNoOfTripleMaps(pn) = 1 then
              if predicate map uses column map then
                SELECT clause ← getColumn(ov)
                trackingListProjectionVariables ← sv, getColumn (ov)
              else if predicate map uses template map then
                SELECT clause ← getColumnFromTemplate (ov)
                trackingListProjectionVariables ← sv, getColumnFromTemplate (ov)
            else if countNoOfTripleMaps(pn) > 1 then
              queueListProjectionVariables ← ov
          end for each
        if pn is rdf:type then
          if ov is identical to one of the class IRIs in subject map of tm then
            if sv in listProjectionVariables then
              SELECT clause ← getColumnFromTemplate (sv)
              trackingListProjectionVariables ← sv, getColumnFromTemplate (sv)
            FROM clause ← bt as tm
            variableTrackingList ← add(sv,bt,tm)
      end for each
    end for each
  end for each
```

3.2.3 Post Processing for FROM Clause

In the previous stage a list of all subject variables that could not be assigned to a triple map unambiguously were added to a queue list. The post processing stage for the FROM clause iterates through this list and establishes whether a combination of the queued subject variable and one of its candidate triple maps, and a variable and triple map combination in the tracking list can be identified. If this is not the case, a match based only on the subject variable is searched for. If a match exists the subject map's template map stem of the matching triple map is compared to the subject map's template map stem of each candidate triple map. If there is a match between two stems, the corresponding triple map for the subject variable is identified. This process is necessary as subclasses can use properties from their parent class. Test query two provides such an example for predicate *foaf:name* used in the last triple pattern. In the example, variable z is the subject for a triple from the actors triple map, but the predicate *foaf:name* is not part of any of the predicate maps in this triple map. However, actors are persons whose triple map contains a predicate map for *foaf:name*.

Pseudo code:

```
for each variable in queueListSubjectVariables
  for each candidate in getTripleMapCandidates(variable)
    if variable and candidate match entry not in variableTrackingList then
      temp ← getTripleMapforVariablefromvariableTrackingList(variable)
      for each candidate
        if getSubjectMapTemplateStem(candidate) = getSubjectMapTemplateStem(temp) then
          bt ← getBaseTableorSqlQuery(temp)
          FROM clause ← bt as temp
          variableTrackingList ← add(variable,bt,temp)
      end for each
  end for each
end for each
```

3.2.4 Post Processing for SELECT Clause

Some of the projection variables might not have been identified unambiguously during the projection processing stage. They were queued and are processed afterwards. In a first step, it is checked whether the projection variable can be found in the projection variable tracking list. If the variable is not found, it is compared against the object variable names stored in the general tracking list. If a match is found, the database column can be identified via the column map as the triple map is already known and the predicate is tracked with every projection variable. If the variable does not match a tracked object variable, the subject variable from the triple pattern that contained the projection variable is used to find the triple map. It is then confirmed whether the triple map found in this way contains the predicate tracked with the projection variable. This allows for the predicate map to be identified and the database column name to be extracted from either a column map or a template map.

Pseudo code:

```
for each (projVariable, subjectVariableProj, predicateProj) in queueListProjectionVariables
  for each projectionVariable in trackingListProjectionVariables
    if projVariable = projectionVariable then
      exit loops
  end for each
  for each (objectVariable, tripleMap) in variableTrackingList
    if objectVariable = projVariable then
      SELECT clause ← getColumnName(tripleMap, predicateProj)
      trackingListProjectionVariables ← projVariable
      exit loops
  end for each
  for each subjectVariable in variableTrackingList
    if subjectVariable = subjectVariableProj then
      tm ← getTripleMap(subjectVariable)
      for each PredicateObjectMap in tm
        if predicateMap contains predicateProj then
          if columnMap then
            SELECT clause ← getColumnName(tm, predicateProj)
          else if templateMap then
            SELECT clause ← getTemplateColumnName(tm, predicateProj)
      end for each
  end for each
end for each
```

3.2.5 EXISTS and NOT EXISTS FILTER Processing

The processing stage for FILTER EXISTS and FILTER NOT EXISTS SPARQL patterns evaluates the triple pattern in the corresponding clause and translates them into selection conditions of a SQL query. This processing stage was implemented with a limited scope. It is assumed that the variable in the subject position is identical to a variable in the subject position of a triple pattern in the main graph pattern. Additionally, the algorithm was only implemented for the case where the object in the FILTER triple pattern is created from a column map and therefore is either a literal or a variable. Object values created from a constant map or via class and template maps are not supported. The triple map that contains the predicate object map with the column map is found via the subject in the FILTER triple pattern as the subject's triple map has been identified at this stage and is recorded in the tracking list. With the triple map known and the predicate from the FILTER triple pattern it is possible to identify the column map and with it the database column name. The selection condition is then created based on the object's type, literal or variable, and the exact FILTER clause, EXISTS or NOT EXISTS. If the object in

the FILTER triple pattern is a variable, and in the list of projection variables, its corresponding database column name is added to the SELECT clause.

Pseudo code:

```
for each exists and not exists filter pattern
  tm ← getTripleMap(subjectVariable)
  for each PredicateObjectMap in tm
    if predicate map contains predicate from filter pattern then
      cn ← getColumnName(tm, predicate from filter pattern)
      if object is literal and EXISTS then
        condition ← getColumn(subjectVariable) = cn
      else if object is literal and NOT EXISTS then
        condition ← getColumn(subjectVariable) != cn
      else if object is variable and EXISTS then
        condition ← cn IS NOT NULL
      else if object is variable and NOT EXISTS then
        condition ← cn IS NULL

      if object is variable and in listProjectionVariables then
        SELECT clause ← cn
  end for each
end for each
```

3.2.6 SELECT Clause Assembly

This stage concatenates all individual SELECT clause components into a comma separated list and adds the SELECT keyword and, if required, the DISTINCT keyword.

3.2.7 FROM Clause Assembly

This stage simply concatenates all individual FROM clause components into a comma separated list and adds the FROM keyword.

3.2.8 WHERE Clause Assembly

The WHERE clause assembly stage analyses the triple patterns in the basic graph pattern as well as FILTER expressions that are not EXISTS or NOT EXISTS patterns and generates corresponding selection conditions for the SQL query.

For achieving the first part, the algorithm iterates through all triple patterns in the main graph. If the object is a variable, its triple map matches the triple map for the subject variable and it is created by a column map, the condition is set to database column name IS NOT NULL. If the triple maps do not match, the object variable is used to identify the triple map related to it. The condition is set to a foreign key join between the database column identified via the template map from the triple pattern's subject variable, and the database column identified from the subject's template map in the triple map found via the object variable. If the object is a literal and created via a column map the condition is determined as a Boolean expression with the database column name equalling the literal.

In the second part, FILTER expressions are translated into corresponding SQL selection conditions. The expressions supported by the query translator application are limited to the comparison operators <, > and = and the regular expressions function. Any variables stated in an expression are translated into

their corresponding database column names. The regular expression function is only translated into the PostgreSQL equivalent and therefore not compatible with other databases.

3.2.9 Finalising SQL Query

In this stage, the different query clauses are combined into one query string and the ORDER BY clause is added if required. The completely translated query is then returned back to the main program.

4 Testing

4.1 Test Cases

The implementation of the query translator is validated through a set of test cases. Each test case addresses a graph pattern or query modifier that was defined as being in scope in section 2.1.3.

A test case is a SPARQL query written on the basis of the RDF dataset that would be generated from the IMDB relational database and its corresponding R2RML mapping if the triples were to be materialised. For performance and readability considerations, eight views containing the records related to ten movies were created and used for the logical table mappings in the R2RML mapping document. Running translated queries against the complete database is possible, but since results are returned as console output, users might not be able to review the translated query displayed above the result list. Some queries might also take longer to run against the complete database. It is therefore recommended to use the views provided with this report. The installation and usage instructions in Appendix A and the definition of the views in Appendix B contain more detailed information.

Table 4.1 gives an overview of the ten test cases and their scope. Each test query was translated with the query translator program and the translated SQL query was then compared with the expected SQL query that was manually translated. The results for each test run are presented in the next section.

Query	Testing scope
Query 1	Basic Graph Pattern over one triple map
Query 2	Basic Graph Pattern over several triple maps and non-unique predicate
Query 3	Distinct clause
Query 4	Order By clause
Query 5	Filter with regex expression
Query 6	Filter with numeric comparison expression
Query 7	Filter with column comparison expression
Query 8	Filter with Exists expression
Query 9	Filter with Not Exists expression
Query 10	Optional triple patterns

Table 4.1: Query test cases and scope

4.2 Test Results

All SPARQL queries used an identical prefix header which is omitted in the following paragraphs. The prefixes are included in the query files provided with the program executables. The exact output of the query translator application is also included in folder "SPARQL test queries translation results" on the data volume accompanying this report.

4.2.1 Query 1

SPARQL query:

```
SELECT ?x ?title ?runtime
WHERE { ?x mo:title     ?title .
        ?x mo:runtime   ?runtime . }
```

Expected SQL query:

```
SELECT MovieTriplesMap.id AS x,
```

```
       MovieTriplesMap.title AS title,
       MovieTriplesMap.runtime AS runtime
FROM   (SELECT * FROM movies) AS MovieTriplesMap
WHERE  MovieTriplesMap.title IS NOT NULL AND
       MovieTriplesMap.runtime IS NOT NULL;
```

Query translator result:

```
SELECT   MovieTriplesMap.id AS x,
         MovieTriplesMap.title AS title,
         MovieTriplesMap.runtime AS runtime
FROM     (SELECT * FROM movies) AS MovieTriplesMap
WHERE    MovieTriplesMap.title IS NOT NULL
AND      MovieTriplesMap.runtime IS NOT NULL
```

Comment:

Test passed.

4.2.2 Query 2

SPARQL query:

```
SELECT ?x ?y ?v ?w ?z
WHERE { ?x mo:title                 ?y .
        ?z mo:isActorIn             ?x .
        ?w dbpedia:performer        ?z .
        ?w rdf:type                 dbpedia:FictionalCharacter .
        ?w foaf:name                "Michael Corleone" .
        ?z foaf:name                ?v . }
```

Expected SQL query:

```
SELECT  MovieTriplesMap.id AS x,
        MovieTriplesMap.title AS y,
        PersonTriplesMap.name AS v,
        CharacterTriplesMap.person_role_id AS w,
        ActorTriplesMap.person_id AS z
FROM    (SELECT * FROM movies) AS MovieTriplesMap,
        (SELECT * FROM persons) AS PersonTriplesMap,
        (SELECT * FROM characters) AS CharacterTriplesMap,
        (SELECT * FROM actors) AS ActorTriplesMap
WHERE   MovieTriplesMap.title IS NOT NULL AND
        ActorTriplesMap.movie id = MovieTriplesMap.id AND
        CharacterTriplesMap.person_id = ActorTriplesMap.person_id AND
        CharacterTriplesMap.name = 'Michael Corleone' AND
        ActorTriplesMap.person_id = PersonTriplesMap.id;
```

Query translator result:

```
SELECT   MovieTriplesMap.id AS x,
         MovieTriplesMap.title AS y,
         PersonTriplesMap.name AS v,
         CharacterTriplesMap.person_role_id AS w,
         ActorTriplesMap.person_id AS z
FROM     (SELECT * FROM movies) AS MovieTriplesMap,
         (SELECT * FROM actors) AS ActorTriplesMap,
         (SELECT * FROM characters) AS CharacterTriplesMap,
         (SELECT * FROM persons) AS PersonTriplesMap
WHERE    MovieTriplesMap.title IS NOT NULL
```

```
AND      ActorTriplesMap.movie_id = MovieTriplesMap.id
AND      CharacterTriplesMap.person_id = ActorTriplesMap.person_id
AND      CharacterTriplesMap.name = 'Michael Corleone'
AND      ActorTriplesMap.person_id = PersonTriplesMap.id
```

Comment:

Test passed.

4.2.3 Query 3

SPARQL query:

```
SELECT DISTINCT ?v
WHERE { ?z mo:isActorIn ?x .
        ?z foaf:name    ?v . }
```

Expected SQL query:

```
SELECT  DISTINCT
        PersonTriplesMap.name AS v
FROM    (SELECT * FROM actors) AS ActorTriplesMap,
        (SELECT * FROM movies) AS MovieTriplesMap,
        (SELECT * FROM persons) AS PersonTriplesMap
WHERE   ActorTriplesMap.movie_id = MovieTriplesMap.id AND
        ActorTriplesMap.person_id = PersonTriplesMap.id AND
        PersonTriplesMap.name IS NOT NULL;
```

Query translator result:

```
SELECT   DISTINCT
         PersonTriplesMap.name AS v
FROM     (SELECT * FROM actors) AS ActorTriplesMap,
         (SELECT * FROM persons) AS PersonTriplesMap
WHERE    ActorTriplesMap.movie_id = ActorTriplesMap.movie_id
AND      ActorTriplesMap.person_id = PersonTriplesMap.id
```

Comment:

Test failed. With exception of a null value, the translated query returns an almost identical set of results as the expected query. However, the query translator failed to include the relation from the MoviesTripleMap triple map and did not include the NOT NULL constraint in the WHERE clause.

The first issue is due to the fact that the translation algorithm cannot find variable x in another triple pattern and therefore relies on the template map in the ActorTriplesMap triple map to find the base table column name. If one were to add the triple pattern `?x rdf:type mo2:Movie` the join on foreign keys would be included in the WHERE clause.

The second issue is caused by the translation algorithm only evaluating each triple pattern once for determining a WHERE condition. In this case, the pattern `?z foaf:name ?v` was used to establish the foreign key join between ActorTriplesMap and PersonTriplesMap. This is an issue that would need to be addressed in a future release.

4.2.4 Query 4

SPARQL query:

```
SELECT ?title ?runtime
WHERE { ?x mo:title           ?title .
        ?x mo:runtime         ?runtime .
        ?x mo:releasedate     ?production_year . }
ORDER BY DESC(?production_year) ASC(?runtime)
```

Expected SQL query:

```
SELECT   MovieTriplesMap.title AS title,
         MovieTriplesMap.runtime AS runtime
FROM     (SELECT * FROM movies) AS MovieTriplesMap
WHERE    MovieTriplesMap.title IS NOT NULL AND
         MovieTriplesMap.runtime IS NOT NULL AND
         MovieTriplesMap.production_year IS NOT NULL
ORDER BY MovieTriplesMap.production_year DESC, MovieTriplesMap.runtime ASC;
```

Query translator result:

```
SELECT   MovieTriplesMap.title AS title,
         MovieTriplesMap.runtime AS runtime
FROM     (SELECT * FROM movies) AS MovieTriplesMap
WHERE    MovieTriplesMap.title IS NOT NULL
AND      MovieTriplesMap.runtime IS NOT NULL
AND      MovieTriplesMap.production_year IS NOT NULL
ORDER BY production_year DESC,
         runtime ASC
```

Comment:

Test passed.

4.2.5 Query 5

SPARQL query:

```
SELECT ?title ?runtime
WHERE { ?x mo:title     ?title .
        ?x mo:runtime   ?runtime .
        FILTER ( regex(?title, "^the godfather", "i") ) }
```

Expected SQL query:

```
SELECT  MovieTriplesMap.title AS title,
        MovieTriplesMap.runtime AS runtime
FROM    (SELECT * FROM movies) AS MovieTriplesMap
WHERE   MovieTriplesMap.title IS NOT NULL AND
        MovieTriplesMap.runtime IS NOT NULL AND
        MovieTriplesMap.title ~* '^the godfather';
```

Query translator result:

```
SELECT   MovieTriplesMap.title AS title,
         MovieTriplesMap.runtime AS runtime
FROM     (SELECT * FROM movies) AS MovieTriplesMap
WHERE    MovieTriplesMap.title IS NOT NULL
AND      MovieTriplesMap.runtime IS NOT NULL
AND      MovieTriplesMap.title ~* '^the godfather'
```

Comment:

Test passed.

4.2.6 Query 6

SPARQL query:

```
SELECT ?title ?production_year
WHERE { ?x mo:title           ?title .
        ?x mo:releasedate     ?production_year .
        FILTER ( ?production_year < 1990 ) }
```

Expected SQL query:

```
SELECT MovieTriplesMap.title AS title,
       MovieTriplesMap.production_year AS production_year
FROM   (SELECT * FROM movies) AS MovieTriplesMap
WHERE  MovieTriplesMap.title IS NOT NULL AND
       MovieTriplesMap.production_year IS NOT NULL AND
       MovieTriplesMap.production_year < 1990;
```

Query translator result:

```
SELECT   MovieTriplesMap.title AS title,
         MovieTriplesMap.production_year AS production_year
FROM     (SELECT * FROM movies) AS MovieTriplesMap
WHERE    MovieTriplesMap.title IS NOT NULL
AND      MovieTriplesMap.production_year IS NOT NULL
AND      MovieTriplesMap.production_year < 1990
```

Comment:

Test passed.

4.2.7 Query 7

SPARQL query:

```
SELECT ?x ?title
WHERE { ?x  a                 mo2:Movie .
        ?x  mo:title          ?title .
        ?x  mo:belongsToGenre mo2:Action .
        ?y  a                 mo2:Movie .
        ?y  mo:title          ?title2 .
        ?y  mo:belongsToGenre mo2:Drama .
        FILTER (?x = ?y ) . }
```

Expected SQL query:

```
SELECT   MovieTriplesMap.id AS x, MovieTriplesMap.title AS title
FROM     (SELECT * FROM movies) AS MovieTriplesMap,
         (SELECT * FROM moviegenres) AS MovieGenreTriplesMap,
         (SELECT * FROM movies) AS MovieTriplesMap2,
         (SELECT * FROM moviegenres) AS MovieGenreTriplesMap2
WHERE    MovieTriplesMap.id = MovieTriplesMap2.id
AND      MovieGenreTriplesMap.genre = 'Action'
AND      MovieGenreTriplesMap2.genre = 'Drama'
AND      MovieTriplesMap.id = MovieGenreTriplesMap.id
AND      MovieTriplesMap2.id = MovieGenreTriplesMap2.id
AND      MovieTriplesMap.title IS NOT NULL
```

```
AND      MovieTriplesMap2.title IS NOT NULL;
```

Query translator result:

```
SELECT   MovieTriplesMap.id AS x,
         MovieGenreTriplesMap.id AS x,
         MovieTriplesMap.title AS title,
         MovieTriplesMap.title AS title2
FROM     (SELECT * FROM movies) AS MovieTriplesMap,
         (SELECT * FROM moviegenres) AS MovieGenreTriplesMap
WHERE    MovieGenreTriplesMap.id = MovieTriplesMap.id
AND      MovieGenreTriplesMap.genre = null
AND      MovieGenreTriplesMap.id = MovieGenreTriplesMap.id
```

Comment:

Test failed. The query translator is not able to evaluate two separate graph patterns. The expression in the FILTER clause is the only link between the two graph patterns. The algorithm would need to rename the identifiers in the FROM clause or create a subquery and join on the primary key column. Time constraints prevented the implementation of such a complex evaluation.

4.2.8 Query 8

SPARQL query:

```
SELECT ?name ?deathdate
WHERE { ?x  a           foaf:Person .
        ?x  foaf:name   ?name .
        FILTER EXISTS {?x dbpedia:deathDate ?deathdate } . }
```

Expected SQL query:

```
SELECT  PersonTriplesMap.name AS name,
        PersonTriplesMap.deathdate AS deathdate
FROM    (SELECT * FROM persons) AS PersonTriplesMap
WHERE   PersonTriplesMap.name IS NOT NULL AND
        PersonTriplesMap.deathdate IS NOT NULL;
```

Query translator result:

```
SELECT   PersonTriplesMap.name AS name,
         PersonTriplesMap.deathdate AS deathdate
FROM     (SELECT * FROM persons) AS PersonTriplesMap
WHERE    PersonTriplesMap.deathdate IS NOT NULL
AND      PersonTriplesMap.name IS NOT NULL
AND      PersonTriplesMap.id = PersonTriplesMap.id
```

Comment:

Test passed. The self-join on the primary key column is added due to the last triple pattern begin added to the post processing queue. Since predicate *foaf:name* is used in two triple maps, the value for the name variable could be established in two ways. An easy solution to this issue is to perform a sanity check on the WHERE clause and removing joins on identical columns.

4.2.9 Query 9

SPARQL query:

```
SELECT ?name ?birthdate
WHERE { ?x  a           foaf:Person .
        ?x  foaf:name   ?name .
        FILTER NOT EXISTS {?x dbpedia:birthDate ?birthdate } . }
```

Expected SQL query:

```
SELECT  PersonTriplesMap.name AS name,
        PersonTriplesMap.birthdate AS birthdate
FROM    (SELECT * FROM persons) AS PersonTriplesMap
WHERE   PersonTriplesMap.name IS NOT NULL AND
        PersonTriplesMap.birthdate IS NULL;
```

Query translator result:

```
SELECT   PersonTriplesMap.name AS name,
         PersonTriplesMap.birthdate AS birthdate
FROM     (SELECT * FROM persons) AS PersonTriplesMap
WHERE    PersonTriplesMap.birthdate IS NULL
AND      PersonTriplesMap.name IS NOT NULL
AND      PersonTriplesMap.id = PersonTriplesMap.id
```

Comment:

Test passed. The same comment as for query eight above applies.

4.2.10 Query 10

SPARQL query:

```
SELECT ?name ?gender ?birthdate ?deathdate
WHERE { ?x  a           foaf:Person .
        ?x  foaf:name   ?name .
        ?x  foaf:gender ?gender .
        OPTIONAL {?x    dbpedia:birthdate ?birthdate } .
        OPTIONAL {?x    dbpedia:deathdate ?deathdate } . }
```

Expected SQL query:

```
SELECT  PersonTriplesMap.name AS name,
        PersonTriplesMap.gender AS gender,
        PersonTriplesMap.birthdate AS birthdate,
        PersonTriplesMap.deathdate AS deathdate
FROM    (SELECT * FROM persons) AS PersonTriplesMap
WHERE   PersonTriplesMap.name IS NOT NULL AND
        PersonTriplesMap.gender IS NOT NULL;
```

Query translator result:

```
SELECT   PersonTriplesMap.name AS name,
         PersonTriplesMap.gender AS gender
FROM     (SELECT * FROM persons) AS PersonTriplesMap
WHERE    PersonTriplesMap.name IS NOT NULL
AND      PersonTriplesMap.id = PersonTriplesMap.id
AND      PersonTriplesMap.gender IS NOT NULL
```

Comment:

Test failed. Time constraints led to the omission of the implementation of the OPTIONAL triple pattern evaluation.

5 Critical Evaluation

The application developed as part of this project and the results presented in this report prove that the translation of SPARQL queries into SQL queries is fairly straightforward, at least for simple queries. The performance of the translation of individual test queries suggests that the processing overhead is not significant as the query translator returned the translated query in a couple of seconds. However, no extensive performance benchmarks were established and it would be interesting to evaluate execution times against more complex and bigger mapping files as the size of the mapping file is one of the main contribution factors to processing time. Other performance metrics could be established by comparing the response times of SPARQL queries against a materialised RDF dataset with the combined timings for translating the queries into equivalent SQL queries and executing them against the relational database. An attempt was made at querying the RDF dataset generated from the IMDB example database by the R2RML Parser application to take such timings. Unfortunately, the Jena SPARQL API was not able to process the RDF dataset due to a parsing error.

In the introduction it was argued that the main purpose of a query translator application was publishing Linked Data directly from a relational database. Currently, the application does not return results in the correct format, for instance JSON or XML, but outputs query results to the console. Despite this being a deliberate choice allowing users to inspect results more conveniently, support for the correct output format would need to be added in a future version for the application to be integrated as a SPARQL endpoint. Another issue related to the integration of the application as a Linked Data interface is that the SPARQL query is read from a file stored at a local path rather than being received by a HTTP request or another network transmission format. These are minor omissions though, that could be addressed in a future release.

A major shortcoming of the translation algorithm is the dramatically reduced scope. However, this was necessary to be able to realistically complete the project in time for submission. It needs to be highlighted that the exclusion of group graph patterns and the failure to implement the translation of OPTIONAL patterns reduced the complexity significantly. Supporting them would require refactoring the current source code, as it currently does not allow generating multiple SQL queries for constructing subquery expressions or perform UNION operations. In general, the code would need to be designed in a more modular way to increase reuse of repetitive operations within the application. In that respect, a more holistic approach to the analysis and design stage would be necessary. The outcome should be a complete specification addressing all SPARQL query operations.

Another area for improvement is the testing approach. For a production-ready system more than ten test queries would need to be defined and applied. Testing should also be extended to databases with different content and structure, and therefore different R2RML mappings.

Apart from the missing support for the complete SPARQL query syntax, there are several other potential enhancements that could be provided in future releases. The support for all major database vendors is one possible addition. Whilst most of the SQL produced by the application is conform to the SQL ISO standard, the regular expression translation is specific to PostgreSQL and would need to be adjusted for other database backends. Another improvement could be the support for entailment regimes. Currently, the query translation algorithm strictly matches RDF terms from the SPARQL query with RDF terms in the mapping document. However, there might be other ontologies defining RDF classes and properties that are not used in the mapping document, but are equivalent to the used ones. Examples are the three properties from the R2RML mapping document used for this project in Table 5.1.

Property	Equivalent properties
mo:runtime	dbpedia:filmRuntime, schema:duration
mo:hasDirector	dbpedia:director, schema:director
mo:isProducedBy	dbpedia:producedBy

Table 5.1: Equivalent Properties

When recalling test query one below, one can see that the query translator is not able to return any results if the property `mo:runtime` were replaced with `dbpedia:filmRuntime`. It is only able to follow simple entailment. If it were to support entailment regimes and the ontology entails the statement `dbpedia:runtime owl:equivalentProperty mo:runtime` , the query translator would return the same results of the query independent of the property used.

```
SELECT ?x ?title ?runtime
WHERE { ?x mo:title     ?title .
        ?x mo:runtime   ?runtime . }
```

Despite all the limitations highlighted above, the application performs as expected considering the scope and produces the desired results with satisfactory speed. It could form the basis for the implementation of the additional SPARQL query features as discussed.

6 Conclusion

This report covers the background, scope and implementation details of the SPARQL to SQL query translation application developed as part of this dissertation project for the MSc in Advanced Computing Technologies.

An introduction to the Semantic Web and its purpose was provided, and the importance of Linked Data to this initiative was highlighted as well. The project's motivation in providing a tool for making relational data available as Linked Data by means of a query translator was outlined and supported by drawing attention to several advantages of this approach.

The report introduced and explained several key concepts and defined the scope of features covering a subset of both the R2RML and SPARQL query specifications. The IMDB relational database schema was presented and utilised as an example and basis for R2RML mapping definitions. The latter made use of several openly available ontologies to describe the relational data in RDF triples.

The main deliverable of this project is the query translation algorithm implemented as a Java application. The rationale of the different stages in the algorithm are explained and detailed in pseudo code. Several test queries are presented and the results of having them translated by the query translator application are included in the report as well as on the accompanying data volume.

The report concludes with the analysis of the impact of scope limitations, a discussion of gaps and issues with the current implementation and highlights possible improvements and new features.

References

Arenas, M., Bertails, A., Prud'hommeaux, E. & Sequeda, J., 2012. *A Direct Mapping of Relational Data to RDF.* [Online]
Available at: http://www.w3.org/TR/rdb-direct-mapping/
[Accessed 23 August 2015].

Berners-Lee, T., Hendler, J. & Lassila, O., 2001. The Semantic Web. *Scientific American*, May, pp. 29-37.

Bizer, C., 2003. *D2R MAP – A Database to RDF Mapping Language.* Budapest, Hungary, Proceedings of the Twelfth International World Wide Web Conference - Posters, WWW 2003.

Bizer, C. & Schultz, A., 2011. The Berlin SPARQL Benchmark. In: A. Sheth, Hrsg. *Semantic Services, Interoperability and Web Applications: Emerging Concepts.* Hershey, PA: IGI Global, pp. 81-103.

Chebotko, A., Lu, S. & Fotouhi, F., 2009. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering,* 68(10), pp. 973-1000.

Chen, Y., Zhao, X. & Zhang, S., 2013. Publishing RDF from Relational Database based on D2R improvement. *WSEAS Transactions on Information Science & Applications,* 10(8), pp. 241-248.

Chong, E. I., Das, S., Eadon, G. & Srinivasan, J., 2005. *An efficient SQL-based RDF Querying Scheme.* Trondheim, Norway, VLDB Endowment.

Das, S., Sundara, S. & Cyganiak, R., 2012. *R2RML: RDB to RDF Mapping Language.* [Online]
Available at: http://www.w3.org/TR/r2rml/
[Zugriff am 15 March 2015].

Elliot, B., Cheng, E., Thomas-Ogbuji, C. & Ozsoyoglu, Z. M., 2009. *A Complete Translation from SPARQL into Efficient SQL.* Cetraro, Italy, ACM New York.

Harris, S. & Seaborne, A., 2013. *SPARQL 1.1 Query Language.* [Online]
Available at: http://www.w3.org/TR/sparql11-query/
[Accessed 23 August 2015].

Heath, T. & Bizer, C., 2011. Linked Data: Evolving the Web into a Global Data Space (1st edition). In: *Synthesis Lecture on the Semantic Web: Theory and Technology.* s.l.:Morgan & Claypool, pp. 1-136.

Hitzler, P. et al., 2012. *OWL 2 Web Ontology Language Primer (Second Edition).* [Online]
Available at: http://www.w3.org/TR/owl2-primer/
[Accessed 23 August 2015].

Kontchakov, R. et al., 2014. *Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime.* Riva del Garda, Italy, Springer International Publishing.

Lv, L., Jiang, H. & Ju, L., 2010. *Research and Implementation of the SPARQL-TO-SQL Query Translation Based on Restrict RDF View.* Sanya, IEEE.

Perez de Laborda, C. & Conrad, S., 2006. *Bringing Relational Data into the SemanticWeb using SPARQL and Relational.OWL.* Atlanta, GA, USA, IEEE.

Perez de Laborda, C., Matthäus, Z. & Stefan, C., 2006. *RDQuery - Querying Relational Databases on-the-fly with RDF-QL.* Podebrady, Czech Republic, Posters and Demos of the 15th International Conference on Knowledge Engineering and Knowledge Mangement, EKAW.

Priyatna, F., Corcho, O. & Sequeda, J., 2014. *Formalisation and Experiences of R2RML-based SPARQL to SQL query translation using Morph.* Seoul, Korea, Proceedings of the 23rd international conference on World wide web, Pages 479-490.

Schreiber, G. & Raimond, Y., 2014. *RDF 1.1 Primer.* [Online]
Available at: http://www.w3.org/TR/rdf11-primer/
[Accessed 23 August 2015].

Sequeda, J. F. & Miranker, D. P., 2013. Ultrawrap: SPARQL Execution on Relational Data. *Web Semantics: Science, Services and Agents on the World Wide Web,* Band 22, pp. 19-39.

Unbehauen, J., Stadler, C. & Auer, S., 2012. Accessing Relational Data on the Web with SparqlMap. In: H. Takeda, Y. Qu, R. Mizoguchi & Y. Kitamura, Hrsg. *Semantic Technology.* Nara, Japan: Springer Berlin Heidelberg, pp. 65-80.

W3C, 2015. *Semantic Web - W3C.* [Online]
Available at: http://www.w3.org/standards/semanticweb/
[Accessed 23 August 2015].

Appendix A: Instructions for using the Software

The query translator application requires that the following software is installed on the client or available on a remote server in case of the Postgres database:

- Java 8 JRE
- PostgreSQL 9.4+

Before the application can be executed, the IMDB database needs to be loaded into the Postgres database. The initial load via the Python tool IMDbPy took 15.5 hours on a laptop with an Intel i7 2.8Ghz CPU, 4GB RAM and a SSD hard drive. In order to enable a speedier import and for convenience, two backup files of the IMDB database are provided on the accompanying DVD in the folder "Database".

1. imdb_bak.backup
2. imdb_small_bak.backup

The first backup file is the complete IMDB database. If it is used to restore the data, the script with name "materialized_views.sql" in the same folder needs to be run against the schema into which the backup was restored. The script creates the SQL views that are referenced by the R2RML mapping document. Alternatively, one can restore these views as base tables in a separate database from the second backup file.

The difference between the two backups is that the first one is the full database for which the views need to be created, whereas the second backup is a small database that only contains the data from the views. It was created with the same SQL queries as the materialized views for the full database.

The following instructions assume that PostgreSQL has been installed and the database service is running.

Restoring the database

1. Start the tool pgAdmin III which is provided as part of the PostgreSQL installation
2. Connect to the database instance
3. Create a new database by right-clicking on Databases in the instance tree view and selecting New Database from the context menu
4. The suggested name for the new database is "imdb"
5. Right-click on the new database in the tree view on the left and select Restore from the context menu
6. Select the backup file to be restored via the filename field and click on Restore
7. The process might take some time if the full IMDB database backup is performed. The message tab will display status updates.
8. (Only required for full IMDB database) Select the database name in the tree view on the left and open the query tool from menu Tools, or hit Ctrl+E. Copy and paste the contents of the file materialized_views.sql into the sql editor window and run all CREATE statements.

Running query translator

1. Copy folder "query_translator-0.1-alpha" in directory "Query Translator Program" on the DVD to a location on the local hard drive.

2. Open file app.properties in a text editor and update the PostgreSQL connection details at the bottom according to the local or remote server
3. Other configuration options apart from the sparql.file parameter should remain unchanged
4. The query translation program can be run by navigating to the folder in a console window and calling "query_translator.bat"
5. The query file containing the SPARQL query that should be translated can be referenced in the app.properties file via the sparql.file parameter.
6. If the program output should be written to a file instead of the console window, one can use the piping command, for example "query_translator.bat > myoutput.txt". The test query outputs generated in this way are stored in directory "SPARQL test queries translation results" on the accompanying DVD.
7. The application also creates a log file named status.log in its root directory which should be inspected for errors.

Appendix B: IMDB Custom Views

The following SQL statements were used to create the views in the IMDB database schema which are referenced as logical tables in the R2RML mapping document. The views will need to be created if the IMDB database is restored in full from the backup file provided. Detailed instructions are provided in Appendix A.

```
-- Movies
DROP MATERIALIZED VIEW movies;
CREATE MATERIALIZED VIEW movies AS
SELECT  CAST(a.id AS INTEGER) AS id,
        a.title,
        CAST(a.production_year AS INTEGER) AS production_year,
        CAST(b.info AS DECIMAL) AS imdbrating,
        CAST(c.info AS INTEGER) AS runtime
FROM    title a,
        movie_info_idx b, movie_info c
WHERE   a.kind_id = 1
AND     a.id = b.movie_id AND b.info_type_id = 101
AND     a.id = c.movie_id AND c.info_type_id = 1
AND     a.id IN
(2089310,2392020,2494129,2790035,2842406,2956281,2971694,2971717,2990074,3018955)
AND     c.info ~ '^\d{1,3}$' AND c.note IS NULL;
CREATE UNIQUE INDEX movieid_idx ON movies (id);

-- MovieGenres
DROP MATERIALIZED VIEW moviegenres;
CREATE MATERIALIZED VIEW moviegenres AS
SELECT  CAST(a.id AS INTEGER) AS id,
        d.info AS genre
FROM    title a
        LEFT OUTER JOIN movie_info d ON a.id = d.movie_id AND d.info_type_id = 3 AND
d.info NOT IN ('Experimental','Short','Erotica','Commercial','Lifestyle')
WHERE   a.kind_id = 1
AND     a.id IN
(2089310,2392020,2494129,2790035,2842406,2956281,2971694,2971717,2990074,3018955);
CREATE INDEX moviegenreid_idx ON moviegenres (id);

-- Person
DROP MATERIALIZED VIEW persons;
CREATE MATERIALIZED VIEW persons AS
SELECT  CAST(a.id AS INTEGER) AS id,
        CASE
         WHEN POSITION(',' IN a.name) > 0 THEN SUBSTRING(a.name FROM POSITION(',' IN
a.name)+2)
         ELSE NULL
        END || ' ' ||
        CASE
         WHEN POSITION(',' IN a.name) > 0 THEN SUBSTRING(a.name FROM 1 FOR
POSITION(',' IN a.name)-1)
         ELSE SUBSTRING(a.name FROM 1)
        END as name,
        CASE
         WHEN POSITION(',' IN a.name) > 0 THEN SUBSTRING(a.name FROM 1 FOR
POSITION(',' IN a.name)-1)
         ELSE SUBSTRING(a.name FROM 1)
        END AS last_name,
        CASE
         WHEN POSITION(',' IN a.name) > 0 THEN SUBSTRING(a.name FROM POSITION(',' IN
a.name)+2)
         ELSE NULL
        END AS first_name,
        CASE
         WHEN a.gender = 'm' THEN 'male'
         WHEN a.gender = 'f' THEN 'female'
         ELSE a.gender
```

```
        END as gender,
        (SELECT to_date(z.info,'dd Month yyyy') FROM person_info z WHERE z.person_id =
a.id AND z.info_type_id = 21
        AND z.info ~ '(\d){1,2} January | February | March | April | May | June | July
| August | September | October | November | December (\d){4}') as birthdate,
        (SELECT to_date(z.info,'dd Month yyyy') FROM person_info z WHERE z.person_id =
a.id AND z.info_type_id = 23
        AND z.info ~ '(\d){1,2} January | February | March | April | May | June | July
| August | September | October | November | December (\d){4}') as deathdate
FROM    name a
WHERE   a.id IN (
                SELECT  b.person_id
                FROM    cast_info b
                WHERE   b.movie_id IN
(2089310,2392020,2494129,2790035,2842406,2956281,2971694,2971717,2990074,3018955)
                AND     ((b.role_id IN (1,2) AND b.nr_order BETWEEN 1 AND 10) OR
(b.role_id IN (8,9)))
                );
CREATE UNIQUE INDEX personid_idx ON persons (id);

-- Actors
DROP MATERIALIZED VIEW actors;
CREATE MATERIALIZED VIEW actors AS
SELECT  CAST(b.person_id AS INTEGER) AS person_id,
        CAST(b.movie_id AS INTEGER) AS movie_id
FROM    cast_info b
WHERE   b.movie_id IN
(2089310,2392020,2494129,2790035,2842406,2956281,2971694,2971717,2990074,3018955)
AND     (b.role_id IN (1,2) AND b.nr_order BETWEEN 1 AND 10);
CREATE INDEX actorid_idx ON actors (person_id, movie_id);

-- Directors
DROP MATERIALIZED VIEW directors;
CREATE MATERIALIZED VIEW directors AS
SELECT  CAST(b.person_id AS INTEGER) AS person_id,
        CAST(b.movie_id AS INTEGER) AS movie_id
FROM    cast_info b
WHERE   b.movie_id IN
(2089310,2392020,2494129,2790035,2842406,2956281,2971694,2971717,2990074,3018955)
AND     b.role_id = 8;
CREATE INDEX directorid_idx ON directors (person_id, movie_id);

-- Editors
DROP MATERIALIZED VIEW editors;
CREATE MATERIALIZED VIEW editors AS
SELECT  CAST(b.person_id AS INTEGER) AS person_id,
        CAST(b.movie_id AS INTEGER) AS movie_id
FROM    cast_info b
WHERE   b.movie_id IN
(2089310,2392020,2494129,2790035,2842406,2956281,2971694,2971717,2990074,3018955)
AND     b.role_id = 9;
CREATE INDEX editorid_idx ON editors (person_id, movie_id);

-- Production Companies
DROP MATERIALIZED VIEW companies;
CREATE MATERIALIZED VIEW companies AS
SELECT  CAST(a.id AS INTEGER) AS id,
        a.name,
        b.movie_id
FROM    company_name a, movie_companies b
WHERE   a.id = b.company_id AND b.company_type_id = 2
        AND b.movie_id IN
(2089310,2392020,2494129,2790035,2842406,2956281,2971694,2971717,2990074,3018955);
CREATE INDEX companyid_idx ON companies (id);
CREATE INDEX companymov_idx ON companies (movie_id);

-- Characters
DROP MATERIALIZED VIEW characters;
CREATE MATERIALIZED VIEW characters AS
```

```
SELECT  CAST(b.person_role_id AS INTEGER) AS person_role_id,
        CAST(b.person_id AS INTEGER) AS person_id,
        CAST(b.movie_id AS INTEGER) AS movie_id,
        d.name
FROM    cast_info b,
        char_name d
WHERE   b.movie_id IN
(2089310,2392020,2494129,2790035,2842406,2956281,2971694,2971717,2990074,3018955)
AND     b.person_role_id = d.id
AND     b.person_id IN (SELECT id FROM persons);
CREATE INDEX characterid_idx ON characters (person_role_id);
```

Appendix C: R2RML Mapping Document

It was necessary to specify logical tables via an R2RML view in the mapping document. Using a schema-qualified table or view name did not work. This seems to be a bug in R2RML Parser.

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix mo: <http://www.movieontology.org/2009/10/01/movieontology.owl#> .
@prefix mo2: <http://www.movieontology.org/2009/11/09/movieontology.owl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dbpedia: <http://dbpedia.org/ontology/> .
@prefix schema: <http://schema.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<#MovieTriplesMap>
    rr:logicalTable
    [
        rr:sqlQuery "SELECT * FROM movies"
    ];

    rr:subjectMap
    [
        rr:template "http://www.imdb.com/movie_{id}";
        rr:class mo2:Movie;
        rr:class dbpedia:Film
    ];

    rr:predicateObjectMap
    [
      rr:predicate      mo:title ;
      rr:objectMap      [ rr:column "title" ]
    ];

    rr:predicateObjectMap
    [
      rr:predicate      mo:releasedate ;
      rr:objectMap      [ rr:column "production_year"; rr:datatype
xsd:integer ]
    ];

    rr:predicateObjectMap
    [
      rr:predicate      mo:imdbrating ;
      rr:objectMap      [ rr:column "imdbrating"; rr:datatype xsd:decimal ]
    ];

    rr:predicateObjectMap
    [
      rr:predicate      mo:runtime ;
      rr:objectMap      [ rr:column "runtime"; rr:datatype xsd:integer ]
    ].

<#MovieGenreTriplesMap>
    rr:logicalTable
    [
        rr:sqlQuery "SELECT * FROM moviegenres"
    ];

    rr:subjectMap
    [
        rr:template "http://www.imdb.com/movie_{id}";
```

```
    ];

    rr:predicateObjectMap
    [
      rr:predicate      rdf:type ;
      rr:objectMap      [ rr:template
"http://www.movieontology.org/2009/10/01/movieontology.owl#Genre" ]
    ];

    rr:predicateObjectMap
    [
      rr:predicate      mo:belongsToGenre ;
      rr:objectMap      [ rr:template
"http://www.movieontology.org/2009/11/09/movieontology.owl#{genre}" ]
    ].

<#PersonTriplesMap>
    rr:logicalTable
    [
        rr:sqlQuery "SELECT * FROM persons"
    ];

    rr:subjectMap
    [
        rr:template "http://www.imdb.com/person {id}";
        rr:class foaf:Person;
    ];

    rr:predicateObjectMap [
      rr:predicate      foaf:name ;
      rr:objectMap      [ rr:column "name" ]
    ];

    rr:predicateObjectMap [
      rr:predicate      foaf:familyName ;
      rr:objectMap      [ rr:column "last_name" ]
    ];

    rr:predicateObjectMap [
      rr:predicate      foaf:givenName ;
      rr:objectMap      [ rr:column "first_name" ]
    ];

    rr:predicateObjectMap [
      rr:predicate      foaf:gender ;
      rr:objectMap      [ rr:column "gender" ]
    ];

    rr:predicateObjectMap [
      rr:predicate      dbpedia:birthDate ;
      rr:objectMap      [ rr:column "birthdate" ]
    ];

    rr:predicateObjectMap [
      rr:predicate      dbpedia:deathDate ;
      rr:objectMap      [ rr:column "deathdate" ]
    ].

<#ActorTriplesMap>
    rr:logicalTable
    [
        rr:sqlQuery "SELECT * FROM actors"
```

```
    ];

    rr:subjectMap
    [
        rr:template "http://www.imdb.com/person {person_id}";
        rr:class dbpedia:Actor;
    ];

    rr:predicateObjectMap [
      rr:predicate      dbpedia:movie ;
      rr:objectMap      [ rr:template "http://www.imdb.com/movie {movie_id}"
]
    ];

    rr:predicateObjectMap [
      rr:predicate      mo:isActorIn ;
      rr:objectMap      [ rr:template "http://www.imdb.com/movie_{movie_id}"
]
    ].

<#DirectorTriplesMap>
    rr:logicalTable
    [
        rr:sqlQuery "SELECT * FROM directors"
    ];

    rr:subjectMap
    [
        rr:template "http://www.imdb.com/person {person_id}";
        rr:class dbpedia:Film_Director;
    ];

    rr:predicateObjectMap [
      rr:predicate      mo:isDirectorOf ;
      rr:objectMap      [ rr:template "http://www.imdb.com/movie_{movie_id}"
]
    ].

<#EditorTriplesMap>
    rr:logicalTable
    [
        rr:sqlQuery "SELECT * FROM editors"
    ];

    rr:subjectMap
    [
        rr:template "http://www.imdb.com/person {person_id}";
        rr:class mo:Editor;
    ];

    rr:predicateObjectMap [
      rr:predicate      mo:isEditorIn ;
      rr:objectMap      [ rr:template "http://www.imdb.com/movie {movie_id}"
]
    ].

<#ProdCompTriplesMap>
    rr:logicalTable
    [
        rr:sqlQuery "SELECT * FROM companies"
    ];
```

```
    rr:subjectMap
    [
        rr:template "http://www.imdb.com/company_{id}";
        rr:class mo:ProductionCompany;
    ];

    rr:predicateObjectMap [
      rr:predicate      schema:legalName ;
      rr:objectMap      [ rr:column "name" ]
    ];

    rr:predicateObjectMap [
      rr:predicate      mo:produced ;
      rr:objectMap      [ rr:template "http://www.imdb.com/movie_{movie_id}"
]
    ].

<#CharacterTriplesMap>
    rr:logicalTable
    [
        rr:sqlQuery "SELECT * FROM characters"
    ];

    rr:subjectMap
    [
        rr:template "http://www.imdb.com/character_{person_role_id}";
        rr:class dbpedia:FictionalCharacter;
    ];

    rr:predicateObjectMap [
      rr:predicate      foaf:name ;
      rr:objectMap      [ rr:column "name" ]
    ];

    rr:predicateObjectMap [
      rr:predicate      dbpedia:movie ;
      rr:objectMap      [ rr:template "http://www.imdb.com/movie_{movie_id}"
]
    ];

    rr:predicateObjectMap [
      rr:predicate      dbpedia:performer ;
      rr:objectMap      [ rr:template
"http://www.imdb.com/person_{person_id}" ]
    ].
```

Appendix D: SPARQL Queries

```
prefix mo:
<http://www.movieontology.org/2009/10/01/movieontology.owl#>
prefix mo2:
<http://www.movieontology.org/2009/11/09/movieontology.owl#>
prefix dbpedia: <http://dbpedia.org/ontology/>
prefix foaf: <http://xmlns.com/foaf/0.1/>

# Query 1: Basic Graph Pattern over one triple map
SELECT ?x ?title ?runtime
WHERE { ?x mo:title     ?title .
        ?x mo:runtime   ?runtime . }

# Query 2: Basic Graph Pattern over several triple maps and non-unique
predicate
SELECT ?x ?y ?v ?w ?z
WHERE { ?x mo:title             ?y .
        ?z mo:isActorIn         ?x .
        ?w dbpedia:performer    ?z .
        ?w rdf:type             dbpedia:FictionalCharacter .
        ?w foaf:name            "Michael Corleone" .
        ?z foaf:name            ?v .
        }

# Query 3: Distinct
SELECT DISTINCT ?v
WHERE { ?z mo:isActorIn ?x .
        ?z foaf:name    ?v .
        }

# Query 4: Order By
SELECT ?title ?runtime
WHERE { ?x mo:title         ?title .
        ?x mo:runtime       ?runtime .
        ?x mo:releasedate   ?production_year . }
ORDER BY DESC(?production_year) ASC(?runtime)

# Query 5: Filter regex
SELECT ?title ?runtime
WHERE { ?x mo:title     ?title .
        ?x mo:runtime   ?runtime .
        FILTER ( regex(?title, "^the godfather", "i") ) }

# Query 6: Filter numeric comparison
SELECT ?title ?production_year
WHERE { ?x mo:title         ?title .
        ?x mo:releasedate   ?production_year .
        FILTER ( ?production_year < 1990 ) }

# Query 7: Filter column comparison
SELECT ?x ?title
WHERE { ?x  a                   mo2:Movie .
        ?x  mo:title            ?title .
        ?x  mo:belongsToGenre   mo2:Action .
        ?y  a                   mo2:Movie .
```

```
        ?y  mo:title            ?title2 .
        ?y  mo:belongsToGenre   mo2:Drama .
        FILTER (?x = ?y ) .}

# Query 8: Filter Exists
SELECT ?name ?deathdate
WHERE { ?x  a           foaf:Person .
        ?x  foaf:name   ?name .
        FILTER EXISTS {?x   dbpedia:deathDate   ?deathdate } .}

# Query 9: Filter Not Exists
SELECT ?name ?birthdate
WHERE { ?x  a           foaf:Person .
        ?x  foaf:name   ?name .
        FILTER NOT EXISTS {?x   dbpedia:birthDate   ?birthdate } .}

# Query 10: Optional
SELECT ?name ?gender ?birthdate ?deathdate
WHERE { ?x  a           foaf:Person .
        ?x  foaf:name   ?name .
        ?x  foaf:gender ?gender .
        OPTIONAL {?x    dbpedia:birthdate   ?birthdate } .
        OPTIONAL {?x    dbpedia:deathdate   ?deathdate } . }
```

Appendix E: Query Translator Java Source Code

The source code written for this project can be found in the directory "Query Translator Source Code\src\main\java\query_translator" on the DVD that was submitted together with this report.

The source code files in this directory have been written from scratch with the following exceptions. The collector classes (CompareCollector, ExistsCollector, FilterClauseCollector, OrderClauseCollector, ProjectionCollector, RegexCollector) used the class StatementPatternCollector[13], which is provided by the Sesame API, as a template.

In addition, the Main class was copied from the R2RML Parser open source code and modified to integrate the QueryTranslator class and remove the Generator class call which creates a file with the materialised RDF dataset. In the Main class file lines 1 to 110 and lines 204 to 213 were copied from R2RML Parser with minor modifications.

Appendix E1: Source Code of Main class

```
public class Main {
	private static final Logger log = LoggerFactory.getLogger(Main.class);

	/**
	 * The properties, as read from the properties file.
	 */
	private static Properties properties = new Properties();

	public static void main(String[] args) {
		Calendar c0 = Calendar.getInstance();
	long t0 = c0.getTimeInMillis();

	CommandLineParser cmdParser = new DefaultParser();

	Options cmdOptions = new Options();
	cmdOptions.addOption("p", "properties", true, "define the properties file. Example:
query_translator -p app.properties");
	cmdOptions.addOption("h", "print help", false, "help");

		String propertiesFile = "app.properties";

		try {
			CommandLine line = cmdParser.parse(cmdOptions, args);

			if (line.hasOption("h")) {
				HelpFormatter help = new HelpFormatter();
				help.printHelp("query_translator\n", cmdOptions);
				System.exit(0);
			}

			if (line.hasOption("p")) {
				propertiesFile = line.getOptionValue("p");
			}
		} catch (ParseException e1) {
			//e1.printStackTrace();
	System.out.println("Error parsing command line arguments.");
			log.error("Error parsing command line arguments.");
			System.exit(1);
		}

		try {
			if (StringUtils.isNotEmpty(propertiesFile)) {
				properties.load(new FileInputStream(propertiesFile));
		System.out.println("Loaded properties from " + propertiesFile);
```

[13] https://bitbucket.org/openrdf/sesame/src/3eb95dcd440e5d4909854c489ca8b48555ff6363/core/queryalgebra/model/src/main/java/org/openrdf/query/algebra/helpers/StatementPatternCollector.java?at=master&fileviewer=file-view-default

```
                                log.info("Loaded properties from " + propertiesFile);
                        }
                } catch (FileNotFoundException e) {
                        //e.printStackTrace();
        System.out.println("Properties file not found (" + propertiesFile + ").");
                        log.error("Properties file not found (" + propertiesFile + ").");
                        System.exit(1);
                } catch (IOException e) {
                        //e.printStackTrace();
        System.out.println("Error reading properties file (" + propertiesFile + ").");
                        log.error("Error reading properties file (" + propertiesFile + ").");
                        System.exit(1);
                }

                ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("app-
context.xml");

                Database db = (Database) context.getBean("db");
                db.setProperties(properties);

                Parser parser = (Parser) context.getBean("parser");
                parser.setProperties(properties);

                MappingDocument mappingDocument = parser.parse();

                mappingDocument.getTimestamps().add(new Long(t0)); //0 Started
                mappingDocument.getTimestamps().add(new Long(Calendar.getInstance().getTimeInMillis()));
//1 Finished parsing. Starting generating result model.

                // Reading SPARQL query file and parsing it
        String queryFile = properties.getProperty("sparql.file");
        InputStream isQuery = FileManager.get().open(queryFile);

        StringWriter writer = new StringWriter();
        try {
                        IOUtils.copy(isQuery, writer, "UTF-8");
                } catch (IOException e1) {
                        // e1.printStackTrace();
        System.out.println("Error reading SPARQL query file.");
                        log.error("Error reading SPARQL query file.");
                        System.exit(1);
                }

        String strQuery = writer.toString();
        SPARQLParser sparqlParser = new SPARQLParser();
        ParsedQuery sparqlQuery = null;

        try {
                        sparqlQuery =
sparqlParser.parseQuery(strQuery,properties.getProperty("default.namespace"));
                } catch (MalformedQueryException e) {
                        // e.printStackTrace();
        System.out.println("Error parsing SPARQL query. Query malformed.");
                        log.error("Error parsing SPARQL query. Query malformed.");
                        System.exit(1);
                }

        // Query translation
        // disallow unsupported query forms
        if (!(sparqlQuery.getTupleExpr().getSignature().equals("Projection") ||
                sparqlQuery.getTupleExpr().getSignature().equals("Distinct"))) {
        System.out.print("SPARQL query form ");
        if (sparqlQuery.getTupleExpr().getSignature().equals("Reduced")) {
                System.out.print("CONSTRUCT");
        }
        else if (sparqlQuery.getTupleExpr().getSignature().equals("DescribeOperator")) {
                System.out.print("DESCRIBE");
        }
        else if (sparqlQuery.getTupleExpr().getSignature().contains("Slice")) {
                System.out.print("ASK");
        }
        else {
                System.out.print(sparqlQuery.getTupleExpr().getSignature());
        }
        System.out.print(" is not supported\r\n\r\n");
```

```
                        log.error("SPARQL query form " + sparqlQuery.getTupleExpr().getSignature() + " is
not supported");
                        System.exit(1);
        }

                // Passing parsed mapping document and SPARQL query to query translator
                String sqlQuery = QueryTranslator.translate(mappingDocument, sparqlQuery);

                // Executing returned sql query and printing results to screen
        log.info("Printing SQL query results to screen");
        System.out.print("\r\n");
        ResultSet rs = db.query(sqlQuery);

        ResultSetMetaData md;
                try {
                        md = rs.getMetaData();
                int colCount = md.getColumnCount();

                for (int i = 1; i <= colCount ; i++) {
                String col_name = md.getColumnName(i);
                System.out.printf("%-50s", col_name);
                }

                System.out.print("\r\n");

                int resultCount = 0;
                while (rs.next()) {
                resultCount++;
                for (int i = 1; i <= colCount; i++) {
                        System.out.printf("%-50s", rs.getString(i));
                }
                System.out.print("\r\n");
                }

                System.out.print("\r\n");
                log.info("Total results found: " + resultCount);
                System.out.print("Total results found: " + resultCount);
                System.out.print("\r\n");
                System.out.print("\r\n");

                } catch (SQLException e) {
                        // e.printStackTrace();
                        log.error("Error printing query results.");
                }

        // ***************************
        // R2RML Parser code continued
        // ***************************
                context.close();
                Calendar c1 = Calendar.getInstance();
        long t1 = c1.getTimeInMillis();
        log.info("Finished in " + (t1 - t0) + " milliseconds.");
        log.info("Done.");
        System.out.println("Done.");
        }

}
```

Appendix E2: Source Code of QueryTranslator class

```
public class QueryTranslator {
        private static final Logger log = LoggerFactory.getLogger(QueryTranslator.class);

        // sets holding individual clause components
        // set is used to avoid duplication of components
        private static Set<String> selectClauseSet;
        private static Set<String> fromClauseSet;
        private static Set<String> whereClauseSet;

        // string variables for assembling individual clauses
        // from elements in above sets
        private static String selectClause;
        private static String fromClause;
        private static String whereClause;
```

```
	// object that tracks subject and object variable, triple map and
	// the base table or SQL view clearly identified for FROM clause
	// for each triple pattern in the SPARQL query
	private static List<List<String>> subVarTrack;
	private static List<List<String>> subVarQueue;

	// object that tracks the variable name, triple map name
	// and base table column name for each projection variable
	private static List<List<String>> projVarTrack;
	private static List<List<String>> projVarQueue;

	// object to track all variables used in triple patterns
	private static List<String> allVarTrack;

	// string variables to hold triple values for each triple pattern
	private static String subjectVar = null;
	private static String predicateIRI = null;
	private static String objectValue = null;

	// string variables to hold triple map name and sql view query
	private static String tripleMapStr;
	private static String tableView;

	// string variable to hold assembled SQL query
	private static String sqlQuery;

	// main translation algorithm
	public static String translate(MappingDocument mappingdoc, ParsedQuery sparqlquery) {

		if(mappingdoc != null && sparqlquery != null) {
			log.info("Initializing translation variables");
			selectClauseSet = new LinkedHashSet<String>();
			fromClauseSet = new LinkedHashSet<String>();
			whereClauseSet = new LinkedHashSet<String>();

			subVarTrack = new ArrayList<List<String>>();
			subVarQueue = new ArrayList<List<String>>();

			projVarTrack = new ArrayList<List<String>>();
			projVarQueue = new ArrayList<List<String>>();

			allVarTrack = new ArrayList<String>();

			sqlQuery = null;

			log.info("Extracting triple patterns from query.");
			// Storing triple maps from mapping documents in list
			LinkedList<LogicalTableMapping> tables = mappingdoc.getLogicalTableMappings();

			// Extract triple patterns in query
		TupleExpr sparqlSelect = sparqlquery.getTupleExpr();
		StatementPatternCollector collector = new StatementPatternCollector();
			sparqlSelect.visit(collector);
			List<StatementPattern> patterns = collector.getStatementPatterns();
			log.info("Extracted triple patterns from query.");

			// Collect projection elements
			log.info("Collecting projection elements.");
		ProjectionCollector visitor = new ProjectionCollector();
			sparqlSelect.visitChildren(visitor);
		List<ProjectionElem> elements = visitor.getProjectionElems();
	if (elements.isEmpty()) {
		log.error("SPARQL query does not contain any projection variables. Operation aborted.");
		System.exit(1);
	}
			log.info("Collected projection elements.");

			log.info("Collecting all variables in query.");
	// Loop through triple patterns in SPARQL query retrieving all variable values
			for (StatementPattern sp : patterns) {
				// get triple variables
				if (!allVarTrack.contains(sp.getSubjectVar().getName())) {
					allVarTrack.add(sp.getSubjectVar().getName());   // subject is
always variable according to scope
```

```
                                }
                                if (!sp.getObjectVar().hasValue() &&
!allVarTrack.contains(sp.getObjectVar().getName())) {
                                        allVarTrack.add(sp.getObjectVar().getName()); // object is
variable
                                }
                        }
                        log.info("Collected all variables in query.");


                        log.info("Starting query translation.");
        // Loop through triple patterns in SPARQL query
                        for (StatementPattern sp : patterns) {
                                // get triple variables
                                subjectVar = sp.getSubjectVar().getName();       // subject is always
variable according to scope
                                predicateIRI = sp.getPredicateVar().getValue().stringValue(); //
predicate is always IRI according to scope
                                // object can be variable, IRI or literal
                                objectValue = null;
                                if (sp.getObjectVar().hasValue()) {
                                        objectValue = sp.getObjectVar().getValue().stringValue(); //
object is IRI or literal
                                        //System.out.println(sp.getObjectVar().getName().indexOf("lit"));

                                        //System.out.println(sp.getObjectVar().getName().indexOf("uri"));

                                }
                                else {
                                        objectValue = sp.getObjectVar().getName(); // object is variable
                                }

                                // identify view in R2RML mapping that generates triple
                        for (LogicalTableMapping ltm : tables) {
                        // Record name of triples map to use it as identifier in FROM clause
                        tripleMapStr = ltm.getUri().substring(ltm.getUri().indexOf("#") + 1);
                        // Record base table or view name or sql query referenced in triple map
                        tableView = ltm.getView().getSelectQuery().getQuery();

                        // if rdf:type or the "a" short hand predicate we need to look in subject map as
well
                        if (predicateIRI.equals("http://www.w3.org/1999/02/22-rdf-syntax-ns#type")) {

                                ArrayList<String> classURIs = ltm.getSubjectMap().getClassUris();

                                for (String uri : classURIs) {
                                        // if class in object matches one of the classes in subjectMap we
have a match
                                                if (sp.getObjectVar().hasValue()) {
                                        if (objectValue.equals(uri)) { // object is an IRI
                                                System.out.print("(" + tableView + ") AS " +
tripleMapStr);
                                                System.out.print(" ==> added by " +
predicateIRI);
                                                System.out.println();
                                                log.debug("(" + tableView + ") AS " +
tripleMapStr + " ==> added by " + predicateIRI);
                                                if (!fromClauseSet.contains("(" + tableView + ")
AS " + tripleMapStr)) {
                                                        fromClauseSet.add("(" + tableView + ")
AS " + tripleMapStr);
                                                        // record in tracking object
                                                        subVarTrack.add(new
ArrayList<String>());
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(subjectVar);
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(tripleMapStr);
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(tableView);
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(objectValue);
                                                }
                                                /*
```

```
                                                                 *              if subject extract column name
from template map
                                                                 *              add column name with triplemap
identifier to select list
                                                                                *
                                                                 */
                                                                for (String varSP : allVarTrack) {
                                                                // projection var equals subject var in triple
pattern
                                                                if (varSP.equals(subjectVar)) {
                                                                        // column(s) referenced in template map
for subject are added to projection var list
                                                                        ArrayList<String> fields =
ltm.getSubjectMap().getTemplate().getFields();

                                                                        for (String field : fields) {
                                                                                // column is prefixed with
triple map name which is used as the table identifier in query
                                                                                // projection variable name is
used as identifier in query
                                                                                System.out.print(tripleMapStr +
"." + field + " AS " + subjectVar);
                                                                                System.out.print(" ==> added by
" + predicateIRI + " line " + Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                                                System.out.println();
                                                                                log.debug(tripleMapStr + "." +
field + " AS " + subjectVar + " ==> added by " + predicateIRI + " line " +
Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                                                selectClauseSet.add(tripleMapStr
+ "." + field + " AS " + subjectVar);
                                                                                        projVarTrack.add(new
ArrayList<String>());

        projVarTrack.get((projVarTrack.size()) - 1).add(subjectVar);

        projVarTrack.get((projVarTrack.size()) - 1).add(tripleMapStr);

        projVarTrack.get((projVarTrack.size()) - 1).add(field);
                                                                        }
                                                                }
                                                                }
                                                        }
                                                                }
                                        }
                        } // if predicate is type close

                        // finding triple map via predicate used in query
                        ArrayList<PredicateObjectMap> pomaps = ltm.getPredicateObjectMaps();
                        for (PredicateObjectMap pomap : pomaps) {

                                ArrayList<String> predicates = pomap.getPredicates();

                                for (String predicate : predicates) {
                                        if (predicateIRI.equals(predicate) &&
!predicate.equals("http://www.w3.org/1999/02/22-rdf-syntax-ns#type")) {
                                                if (noOfMatchesTM(predicate, mappingdoc) == 1){
                                                        // we can safely add triple map view to FROM
clause
                                                        System.out.print("(" + tableView + ") AS " +
tripleMapStr);
                                                        System.out.print(" ==> added by " +
predicateIRI);
                                                        System.out.println();
                                                        log.debug("(" + tableView + ") AS " +
tripleMapStr + " ==> added by " + predicateIRI);
                                                        if (!fromClauseSet.contains("(" + tableView + ")
AS " + tripleMapStr)) {
                                                                fromClauseSet.add("(" + tableView + ")
AS " + tripleMapStr);
                                                                // record in tracking object
                                                                subVarTrack.add(new
ArrayList<String>());
                                                                subVarTrack.get((subVarTrack.size()) -
1).add(subjectVar);
                                                                subVarTrack.get((subVarTrack.size()) -
1).add(tripleMapStr);
```

```
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(tableView);
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(objectValue);
                                                }
                                                // we can also safely add the subject variable to
the SELECT clause,
                                                // provided it is in the subset of projection
variables
                                                        for (String varSP : allVarTrack) {
                                                        if (varSP.equals(subjectVar)) {
                                                                // column(s) referenced in
template map for subject are added to projection var list
                                                                ArrayList<String> fields =
ltm.getSubjectMap().getTemplate().getFields();

                                                                for (String field : fields) {
                                                                        // column is prefixed
with triple map name which is used as the table identifier in query
                                                                        // projection variable
name is used as identifier in query
                                                                System.out.print(tripleMapStr +
"." + field + " AS " + subjectVar);
                                                                System.out.print(" ==> added by
" + predicateIRI + " line " + Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                                System.out.println();
                                                                log.debug(tripleMapStr + "." +
field + " AS " + subjectVar + " ==> added by " + predicateIRI + " line " +
Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                                selectClauseSet.add(tripleMapStr
+ "." + field + " AS " + subjectVar);
                                                                        projVarTrack.add(new
ArrayList<String>());
        projVarTrack.get((projVarTrack.size()) - 1).add(subjectVar);
        projVarTrack.get((projVarTrack.size()) - 1).add(tripleMapStr);
        projVarTrack.get((projVarTrack.size()) - 1).add(field);
                                                                }
                                                        }
                                                        }
                                                }
                                                // there is more than one option
                                                else {
                                                        // check if there is a match for the combination
of subject and one of the options
                                                        List<String> tmMatches = matchingTM(predicate,
mappingdoc);
                                                        boolean tmexists = false;

                                                        for (int i=0; i < subVarTrack.size(); i++) {
                                                                for (String tmMatch : tmMatches) {
                                                                        if
(subVarTrack.get(i).get(0).equals(subjectVar)
                                                                                        &&
subVarTrack.get(i).get(1).equals(tmMatch)) {
                                                                        tmexists = true;
                                                                        System.out.println("The
subject var " + subjectVar + " was added with " + tmMatch);
                                                                log.debug("The subject var " +
subjectVar + " was added with " + tmMatch);
                                                                }

                                                        }
                                                }

                                                if (!tmexists) {
                                                        // if not queue for post processing
                                                        System.out.println("Subject var " +
subjectVar + " and predicate " + predicate + " queued for post-processing");
                                                log.debug("Subject var " + subjectVar + " and
predicate " + predicate + " queued for post-processing");
                                                        // record in queue object
```

```
                                                        subVarQueue.add(new
ArrayList<String>());
                                                        subVarQueue.get((subVarQueue.size()) -
1).add(subjectVar);
                                                        subVarQueue.get((subVarQueue.size()) -
1).add(predicate);
                                                        subVarQueue.get((subVarQueue.size()) -
1).add(objectValue);
                                                }
                                        }
                                        // loop through projection variables to check which
position they are occurring in the triple
                                                for (String varSP : allVarTrack) {
                                                // check whether projection variable has been
identified already
                                                boolean projexists = false;
                                                for (int i=0; i < projVarTrack.size(); i++) {
                                                        if
(projVarTrack.get(i).get(0).equals(varSP)) {
                                                                projexists = true;
                                                                //System.out.println("The
projection var " + projVarTrack.get(i).get(0) + " was identified as " + projVarTrack.get(i).get(1) + "." +
projVarTrack.get(i).get(2));
                                                                }
                                                        }
                                                if (!projexists) {
                                                        // check if projection variable is in
subject position
                                                        if (varSP.equals(subjectVar)) {
                                                                // column(s) referenced in
template map for subject are added to projection var list
                                                                ArrayList<String> fields =
ltm.getSubjectMap().getTemplate().getFields();

                                                                for (String field : fields) {
                                                                        // column is prefixed
with triple map name which is used as the table identifier in query
                                                                        // projection variable
name is used as identifier in query
                                                                System.out.print(tripleMapStr +
"." + field + " AS " + subjectVar);
                                                                System.out.print(" ==> added by
" + predicateIRI + " line " + Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                                System.out.println();
                                                                log.debug(tripleMapStr + "." +
field + " AS " + subjectVar + " ==> added by " + predicateIRI + " line " +
Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                                selectClauseSet.add(tripleMapStr
+ "." + field + " AS " + subjectVar);
                                                                        projVarTrack.add(new
ArrayList<String>());
        projVarTrack.get((projVarTrack.size()) - 1).add(subjectVar);
        projVarTrack.get((projVarTrack.size()) - 1).add(tripleMapStr);
        projVarTrack.get((projVarTrack.size()) - 1).add(field);
                                                                }
                                                        }
                                                        // checking if projection variable is in
object position
                                                        // if yes add column field to select
list
                                                        else if (varSP.equals(objectValue)) {
                                                                // there is only one matching
PredicateObjectMap
                                                                if (noOfMatchesTM(predicateIRI,
mappingdoc) == 1) {
                                                                        // first look for column
map and if exists use column value
                                                                        if
(pomap.getObjectColumn() != null) {
        System.out.print(tripleMapStr + "." + pomap.getObjectColumn() + " AS " + objectValue);
```

```
                                                                        System.out.print(" ==>
added by " + predicateIRI + " line " + Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                                        System.out.println();
                                                                        log.debug(tripleMapStr +
"." + pomap.getObjectColumn() + " AS " + objectValue + " ==> added by " + predicateIRI + " line " +
Thread.currentThread().getStackTrace()[1].getLineNumber());

        selectClauseSet.add(tripleMapStr + "." + pomap.getObjectColumn() + " AS " + objectValue);

        projVarTrack.add(new ArrayList<String>());

        projVarTrack.get((projVarTrack.size()) - 1).add(objectValue);

        projVarTrack.get((projVarTrack.size()) - 1).add(tripleMapStr);

        projVarTrack.get((projVarTrack.size()) - 1).add(pomap.getObjectColumn());


        //whereClauseSet.add(tripleMapStr + "." + pomap.getObjectColumn() + " IS NOT NULL");
                                                                        }
                                                                        // if no column map,
must be template map
                                                                        else {
                                                                                // extract
column value from template map

        ArrayList<String> fields = pomap.getObjectTemplate().getFields();

                                                                                for (String
field : fields) {

        System.out.print(tripleMapStr + "." + field + " AS " + objectValue);

        System.out.print(" ==> added by " + predicateIRI + " line " +
Thread.currentThread().getStackTrace()[1].getLineNumber());

        System.out.println();

        log.debug(tripleMapStr + "." + field + " AS " + objectValue + " ==> added by " + predicateIRI + "
line " + Thread.currentThread().getStackTrace()[1].getLineNumber());

        selectClauseSet.add(tripleMapStr + "." + field + " AS " + objectValue);

        projVarTrack.add(new ArrayList<String>());

        projVarTrack.get((projVarTrack.size()) - 1).add(objectValue);

        projVarTrack.get((projVarTrack.size()) - 1).add(tripleMapStr);

        projVarTrack.get((projVarTrack.size()) - 1).add(field);

                                                                                }
                                                                        }

                                                                }
                                                                // there is more than one
PredicateObjectMap
                                                                else {
                                                                        // record in queue
object
                                                                        if
(!projVarQueue.contains(varSP)) {
                                                                                // add to queue
for post-processing

        System.out.println("Projection var " + varSP + " queued for post-processing");
                                                                        log.debug("Projection
var " + varSP + " queued for post-processing");

        projVarQueue.add(new ArrayList<String>());

        projVarQueue.get((projVarQueue.size()) - 1).add(varSP);

        projVarQueue.get((projVarQueue.size()) - 1).add(predicateIRI);
```

```
        projVarQueue.get((projVarQueue.size()) - 1).add(subjectVar);
                                                                                }
                                                                        }
                                                                }
                                                        }
                                                        }
                                        }
                                        else if (predicate.equals("http://www.w3.org/1999/02/22-rdf-
syntax-ns#type")) {
                                                // if class in object matches the class related via
ObjectMap we have a match
                                                                if (sp.getObjectVar().hasValue()) {
                                                        if
(objectValue.equals(pomap.getObjectTemplate().getText())) {
                                                                System.out.print("(" + tableView + ") AS
" + tripleMapStr);
                                                                System.out.print(" ==> added by " +
predicateIRI);
                                                                System.out.println();
                                                        log.debug("(" + tableView + ") AS " +
tripleMapStr + " ==> added by " + predicateIRI);
                                                                if (!fromClauseSet.contains("(" +
tableView + ") AS " + tripleMapStr)) {
                                                                        fromClauseSet.add("(" +
tableView + ") AS " + tripleMapStr);
                                                                        // record in tracking object
                                                                        subVarTrack.add(new
ArrayList<String>());
        subVarTrack.get((subVarTrack.size()) - 1).add(subjectVar);
        subVarTrack.get((subVarTrack.size()) - 1).add(tripleMapStr);
        subVarTrack.get((subVarTrack.size()) - 1).add(tableView);
        subVarTrack.get((subVarTrack.size()) - 1).add(objectValue);
                                                                }
                                                                for (String varSP : allVarTrack) {
                                                                // projection var equals subject var in
triple pattern
                                                                if (varSP.equals(subjectVar)) {
                                                                        // column(s) referenced in
template map for subject are added to projection var list
                                                                        ArrayList<String> fields =
ltm.getSubjectMap().getTemplate().getFields();
                                                                        for (String field : fields) {
                                                                                // column is prefixed
with triple map name which is used as the table identifier in query
                                                                                // projection variable
name is used as identifier in query
        System.out.print(tripleMapStr + "." + field + " AS " + subjectVar);
                                                                                System.out.print(" ==>
added by " + predicateIRI + " line " + Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                                                System.out.println();
                                                                                log.debug(tripleMapStr +
"." + field + " AS " + subjectVar + " ==> added by " + predicateIRI + " line " +
Thread.currentThread().getStackTrace()[1].getLineNumber());
        selectClauseSet.add(tripleMapStr + "." + field + " AS " + subjectVar);
        projVarTrack.add(new ArrayList<String>());
        projVarTrack.get((projVarTrack.size()) - 1).add(subjectVar);
        projVarTrack.get((projVarTrack.size()) - 1).add(tripleMapStr);
        projVarTrack.get((projVarTrack.size()) - 1).add(field);
                                                                        }
                                                                }
                                                                }
                                                        }
```

```
                                                                }
                                                        }
                                                } // ltm predicate loop close
                                        } // pom loop close
                                        }       //      ltm loop close
                                        } // statement patterns loop close

                                        // post processing for FROM clause
                                        // load queue
                                        for (int i = 0; i < subVarQueue.size(); i++) {
                                                List<String> tmMatches = matchingTM(subVarQueue.get(i).get(1),
mappingdoc);
                                                boolean tmexists = false;
                                                for (String tmMatch : tmMatches) {
                                                        // check against items in tracking list
                                                        for (int j = 0; j < subVarTrack.size(); j++) {
                                                                if
(subVarQueue.get(i).get(0).equals(subVarTrack.get(j).get(0))
                                                                                &&
tmMatch.equals(subVarTrack.get(j).get(1))) {
                                                                        tmexists = true;
                                                                        System.out.println("The subject var " +
subVarQueue.get(i).get(0) + " was added with " + tmMatch);
                                                                log.debug("The subject var " + subVarQueue.get(i).get(0) + " was
added with " + tmMatch);
                                                                }
                                                        }
                                                }

                                                // if no matches, compare subject template stem of options with subject
template stem of triple map associated with subject
                                                if (!tmexists) {
                                                        System.out.println("No matching triple map found for variable " +
subVarQueue.get(i).get(0));
                                                        log.debug("No matching triple map found for variable " +
subVarQueue.get(i).get(0));
                                                        String subjectTM = "";
                                                        // find the subject variable in tracking object and extract
triple map
                                                        // this triple map can be used to find column
                                                        for (int j = 0; j < subVarTrack.size(); j++) {
                                                                if
(subVarQueue.get(i).get(0).equals(subVarTrack.get(j).get(0))) {
                                                                        subjectTM = subVarTrack.get(j).get(1);
                                                                        System.out.println("The subject var " +
subVarQueue.get(i).get(0) + " has a related TM " + subjectTM);
                                                                log.debug("The subject var " + subVarQueue.get(i).get(0) + " has
a related TM " + subjectTM);
                                                                }
                                                        }

                                                        if (subjectTM != null) {
                                                                // check if template stems match for related TM and one
of the matches from complete LTM
                                                                for (String tmMatch : tmMatches) {
                                                                        String queryText = getTMSubjectTemplate(tmMatch,
mappingdoc).getView().getSelectQuery().getQuery();
                                                                        if (getTMSubjectTemplate(tmMatch,
mappingdoc).getSubjectMap().getTemplate().getText().replaceAll("\\{.[a-zA-Z_]+\\}",
"").equals(getTMSubjectTemplate(subjectTM,
mappingdoc).getSubjectMap().getTemplate().getText().replaceAll("\\{.[a-zA-Z_]+\\}", ""))) {
                                                                                // add combination to tracking object
and to FROM clause
                                                                        System.out.print("(" + queryText + ") AS " + tmMatch);

                                                                        System.out.print(" ==> added by " +
subVarQueue.get(i).get(2));
                                                                        System.out.println();
                                                                        log.debug("(" + queryText + ") AS " + tmMatch + " ==>
added by " + subVarQueue.get(i).get(2));
                                                                        if (!fromClauseSet.contains("(" + queryText + ") AS " +
tmMatch)) {
                                                                                fromClauseSet.add("(" + queryText + ") AS " +
tmMatch);
                                                                                // record in tracking object
```

```
                                                        subVarTrack.add(new ArrayList<String>());
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(subVarQueue.get(i).get(0));
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(tmMatch);
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(queryText);
                                                        subVarTrack.get((subVarTrack.size()) -
1).add(subVarQueue.get(i).get(2));
                                                }
                                                        }
                                                }
                                        }
                                        else {
                                                // if still no matches --> no triple map can be
identified reliably
                                                log.error("Not able to determine Triple Map for subject
variable " + subVarQueue.get(i).get(0));
                                        }
                                }
                        }

                        // post-processing for SELECT clause
                        for (int i = 0; i < projVarQueue.size(); i++) {
                                boolean projexists = false;
                                // first look in tracking list
                                for (int j = 0; j < projVarTrack.size(); j++) {
                                        if
(projVarQueue.get(i).get(0).equals(projVarTrack.get(j).get(0))) {
                                                projexists = true;
                                                System.out.println("The projection var " +
projVarQueue.get(i).get(0) + " was identified as " + projVarTrack.get(j).get(1) + "." +
projVarTrack.get(j).get(2));
                                                log.debug("The projection var " + projVarQueue.get(i).get(0) + " was
identified as " + projVarTrack.get(j).get(1) + "." + projVarTrack.get(j).get(2));
                                        }
                                }

                                if (!projexists) {
                                        String columnName = null;
                                        // Compare against object value in tracking list
                                        for (int j = 0; j < subVarTrack.size(); j++) {
                                                // if found extract column name from PredicateObjectMap
                                                if
(projVarQueue.get(i).get(0).equals(subVarTrack.get(j).get(3))) {
                                                        columnName =
getColumnName(subVarTrack.get(j).get(1), projVarQueue.get(i).get(1), mappingdoc);
                                                if (columnName != null) {

                                                System.out.print(subVarTrack.get(j).get(1) + "." +
columnName + " AS " + projVarQueue.get(i).get(0));
                                                System.out.print(" ==> added by " +
projVarQueue.get(i).get(1) + " line " + Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                System.out.println();
                                                log.debug(subVarTrack.get(j).get(1) + "." + columnName +
" AS " + projVarQueue.get(i).get(0) + " ==> added by " + projVarQueue.get(i).get(1) + " line " +
Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                selectClauseSet.add(subVarTrack.get(j).get(1) + "." +
columnName + " AS " + projVarQueue.get(i).get(0));
                                                projVarTrack.add(new ArrayList<String>());
                                                projVarTrack.get((projVarTrack.size()) -
1).add(projVarQueue.get(i).get(0));
                                                projVarTrack.get((projVarTrack.size()) -
1).add(subVarTrack.get(j).get(1));
                                                projVarTrack.get((projVarTrack.size()) -
1).add(columnName);
                                        }
                                        else {
                                                                System.out.println("The projection var "
+ projVarQueue.get(i).get(0) + " could not be identified");
                                                                log.error("The projection var " +
projVarQueue.get(i).get(0) + " could not be identified");
                                        }
                                                }
```

```
                                                // otherwise check whether subject's triple map
identified and whether
                                                // that triple map has the predicate relating to the
variable
                                                // if yes then use column map or template map to extract
column name
                                                else if
(projVarQueue.get(i).get(2).equals(subVarTrack.get(j).get(0))) {
                                                        columnName =
getColumnName(subVarTrack.get(j).get(1), projVarQueue.get(i).get(1), mappingdoc);
                                                if (columnName != null) {

                                                System.out.print(subVarTrack.get(j).get(1) + "." +
columnName + " AS " + projVarQueue.get(i).get(0));
                                                System.out.print(" ==> added by " +
projVarQueue.get(i).get(1) + " line " + Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                System.out.println();
                                                log.debug(subVarTrack.get(j).get(1) + "." + columnName +
" AS " + projVarQueue.get(i).get(0) + " ==> added by " + projVarQueue.get(i).get(1) + " line " +
Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                selectClauseSet.add(subVarTrack.get(j).get(1) + "." +
columnName + " AS " + projVarQueue.get(i).get(0));
                                                projVarTrack.add(new ArrayList<String>());
                                                projVarTrack.get((projVarTrack.size()) -
1).add(projVarQueue.get(i).get(0));
                                                projVarTrack.get((projVarTrack.size()) -
1).add(subVarTrack.get(j).get(1));
                                                projVarTrack.get((projVarTrack.size()) -
1).add(columnName);
                                        }
                                        else {
                                                        System.out.println("The projection var "
+ projVarQueue.get(i).get(0) + " could not be identified");
                                                        log.error("The projection var " +
projVarQueue.get(i).get(0) + " could not be identified");
                                        }
                                                }
                                                else {
                                                        System.out.println("The projection var " +
projVarQueue.get(i).get(0) + " could not be identified");
                                                        log.error("The projection var " +
projVarQueue.get(i).get(0) + " could not be identified");
                                                }
                                        }
                                }
                        }

                        // Evaluating Filter Exists and Filter Not Exists clauses
                ExistsCollector existsVisitor = new ExistsCollector();
                        sparqlSelect.visitChildren(existsVisitor);
                List<Exists> exists = existsVisitor.getExists();
                if (!exists.isEmpty()) {
                        for (Exists exist : exists) {
                        StatementPatternCollector filterpattern = new StatementPatternCollector();
                        exist.visit(filterpattern);
                                List<StatementPattern> filterpatterns =
filterpattern.getStatementPatterns();
                                for (StatementPattern filter : filterpatterns) {

                                        // Case 1: object related via constant map
                                                // TODO: not implemented

                                        // Case 2: object is IRI
                                        // Case 2a: object related via template map
                                        // Case 2b: object related via rdf:type
                                                // TODO: not implemented

                                        // Case 3: object related via column map
                                        // we assume subject is identical to one triple in graph pattern
                                        // therefore we can find triple map based on subject
                                        if
(!filter.getPredicateVar().getValue().stringValue().equals("http://www.w3.org/1999/02/22-rdf-syntax-
ns#type")) {
                                                String columnName = null;
                                                String tripleMap = null;
```

```
                                                        for (int j = 0; j < subVarTrack.size(); j++) {
                                                                if
(filter.getSubjectVar().getName().equals(subVarTrack.get(j).get(0))) {
                                                                        columnName =
getColumnName(subVarTrack.get(j).get(1), filter.getPredicateVar().getValue().stringValue(), mappingdoc);
                                                                        tripleMap =
subVarTrack.get(j).get(1);
                                                                }
                                                        }
                                                        if (columnName != null) {
                                                                // if object is literal
                                                                if (filter.getObjectVar().hasValue() &&
filter.getObjectVar().getName().indexOf("lit") > 0) {
                                                        if (filter.getParentNode().getParentNode()
instanceof Not) {
                                                                //System.out.println("NOT EXISTS");

        whereClauseSet.add(tripleMap + "." + columnName + " != '" +
filter.getObjectVar().getValue().stringValue() + "'");
                                                        }
                                                        else if (filter.getParentNode().getParentNode()
instanceof Filter) {
                                                                //System.out.println("EXISTS");

        whereClauseSet.add(tripleMap + "." + columnName + " = '" +
filter.getObjectVar().getValue().stringValue() + "'");
                                                        }
                                                                }
                                                                // object is IRI
                                                                else if
(filter.getObjectVar().hasValue() && filter.getObjectVar().getName().indexOf("uri") > 0) {
                                                                        // TODO: not implemented
                                                                }
                                                                // if object is variable
                                                                else {
                                                        if (filter.getParentNode().getParentNode()
instanceof Not) {
                                                                //System.out.println("NOT EXISTS");

        whereClauseSet.add(tripleMap + "." + columnName + " IS NULL");
                                                        }
                                                        else if (filter.getParentNode().getParentNode()
instanceof Filter) {
                                                                //System.out.println("EXISTS");

        whereClauseSet.add(tripleMap + "." + columnName + " IS NOT NULL");
                                                        }

                                                        System.out.print(tripleMap + "." + columnName + "
AS " + filter.getObjectVar().getName());
                                                        System.out.print(" ==> added by " +
filter.getPredicateVar().getValue().stringValue() + " line " +
Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                        System.out.println();
                                                        log.debug(tripleMap + "." + columnName + " AS " +
filter.getObjectVar().getName() + " ==> added by " + filter.getPredicateVar().getValue().stringValue() + "
line " + Thread.currentThread().getStackTrace()[1].getLineNumber());
                                                                selectClauseSet.add(tripleMap +
"." + columnName + " AS " + filter.getObjectVar().getName());
                                                        projVarTrack.add(new ArrayList<String>());
                                                        projVarTrack.get((projVarTrack.size()) -
1).add(filter.getObjectVar().getName());
                                                        projVarTrack.get((projVarTrack.size()) -
1).add(tripleMap);
                                                        projVarTrack.get((projVarTrack.size()) -
1).add(columnName);
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }

                        // putting the SELECT clause together
                        log.info("Assembling SELECT clause");
```

```
                        selectClause = "";
                        // we need to sort the projection clause according to
                        // the order of projection elements in SPARQL query
                        for (ProjectionElem projElement : elements) {
                                for (String str : selectClauseSet) {
                                        if (str != null && str.contains("AS " +
projElement.getSourceName())) {
                                                selectClause += str + ", ";

                                        }
                                }
                        }
                        if (sparqlquery.getTupleExpr().getSignature().equals("Distinct")) {
                                selectClause = "SELECT DISTINCT " + selectClause.substring(0,
selectClause.length() - 2);
                        }
                        else {
                                selectClause = "SELECT " + selectClause.substring(0,
selectClause.length() - 2);
                        }

                        // putting the FROM clause together
                        Log.info("Assembling FROM clause");
                        fromClause = "";
                        for (String str : fromClauseSet) {
                                if (str != null) {
                                        fromClause += str + ", ";

                                }
                        }
                        fromClause = "FROM " + fromClause.substring(0, fromClause.length() - 2);

                        // putting the WHERE clause together
                        Log.info("Assembling WHERE clause");
                        for (StatementPattern sp : patterns) {
                                // get triple variables
                                subjectVar = sp.getSubjectVar().getName();       // subject is always
variable according to scope
                                predicateIRI = sp.getPredicateVar().getValue().stringValue(); //
predicate is always IRI according to scope
                                // object can be variable, IRI or literal
                                objectValue = null;
                                if (sp.getObjectVar().hasValue()) {
                                        objectValue = sp.getObjectVar().getValue().stringValue(); //
object is IRI or literal
                                        //System.out.println(sp.getObjectVar().getName().indexOf("lit"));

                                        //System.out.println(sp.getObjectVar().getName().indexOf("uri"));

                                }
                                else {
                                        objectValue = sp.getObjectVar().getName(); // object is variable
                                }

                                // check whether more than one candidate triple map
//                              if (noOfMatchesTM(predicateIRI, mappingdoc) == 1) {

                                        // get triple map already identified for subject variable
                                        String idTripleMap = null;
                                        for (int j = 0; j < projVarTrack.size(); j++) {
                                                if (subjectVar.equals(projVarTrack.get(j).get(0))) {
                                                        idTripleMap = projVarTrack.get(j).get(1);
                                                }
                                        }

                                for (LogicalTableMapping ltm : tables) {
                                tripleMapStr = ltm.getUri().substring(ltm.getUri().indexOf("#") + 1);

                                //if (idTripleMap.equals(tripleMapStr)) {

                                        ArrayList<PredicateObjectMap> pomaps =
ltm.getPredicateObjectMaps();
                                        for (PredicateObjectMap pomap : pomaps) {

                                                ArrayList<String> predicates = pomap.getPredicates();
```

```
                                                for (String predicate : predicates) {
                                                        if (predicateIRI.equals(predicate)) {
                                                                // if object is a variable and created
via a column map, set condition to column IS NOT NULL
                                                                // only if triple map for subject and
object variables match
                                                                if (pomap.getObjectColumn() != null &&
!sp.getObjectVar().hasValue()) {
                                                                        if
(getTMforVar(subjectVar).equals(getTMforVar(objectValue)) && idTripleMap.equals(tripleMapStr)) { // triple
map for subject and object variable match

        whereClauseSet.add(tripleMapStr + "." + pomap.getObjectColumn() + " IS NOT NULL");
                                                                        }
                                                                        // if not use subject column
from object related TM and join on subject column from triple pattern
                                                                        else { // no match, join on
foreign key

        whereClauseSet.add(getColumn(subjectVar) + " = " +

        getTMforVar(objectValue) + "." + getTMSubjectTemplate(getTMforVar(objectValue),
mappingdoc).getSubjectMap().getTemplate().getFields().get(0));
                                                                        }
                                                                }
                                                                // if object is a literal and created
via a column map, set condition to column = literal
                                                                else if (pomap.getObjectColumn() != null
&& sp.getObjectVar().hasValue()
                                                                                &&
sp.getObjectVar().getName().indexOf("lit") > 0 && idTripleMap.equals(tripleMapStr)) {
                                                                        whereClauseSet.add(tripleMapStr
+ "." + pomap.getObjectColumn() + " = " + "'" + sp.getObjectVar().getValue().stringValue() + "'");
                                                                }
                                                                // if no column map, must be template
map
                                                                // set condition to subject column =
object column
                                                                else if
(idTripleMap.equals(tripleMapStr)) {
                                                                        // extract column value from
template map
                                                                        ArrayList<String> fields =
pomap.getObjectTemplate().getFields();

                                                                        // scope defines that there will
be only one field, list has maximum length of one
                                                                        for (String field : fields) {

        whereClauseSet.add(tripleMapStr + "." + field + " = " + getColumn(sp.getObjectVar().getName()));
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                //}
                                }
//                              }
//                              // more than one triple map
//                              else {
//                              }

                        }
                        whereClause = "";
                        for (String str : whereClauseSet) {
                                if (str != null) {
                                        whereClause += str + " AND ";

                                }
                        }

                        if (!evalFilter(sparqlSelect, projVarTrack).isEmpty()) {
                                for (String str : evalFilter(sparqlSelect, projVarTrack)) {
                                        if (str != null) {
```

```
                                        whereClause += str + " AND ";

                                }
                        }
                }

                if (whereClause != null && whereClause != "") {
                        whereClause = "WHERE " + whereClause.substring(0, whereClause.length() -
4);
                }

                // putting it all together
                log.info("Assembling complete SQL query");
                sqlQuery = (whereClause != null) ? selectClause + " " + fromClause + " " +
whereClause : selectClause + " " + fromClause;

                // adding ORDER BY clause if required
                log.info("Adding ORDER BY clause");
        OrderClauseCollector orderVisitor = new OrderClauseCollector();
                sparqlSelect.visitChildren(orderVisitor);
        List<OrderElem> orderElements = orderVisitor.getOrderElems();
        Var orderVar = null;
        if (!orderElements.isEmpty()) {
                sqlQuery = sqlQuery + "ORDER BY ";

                for (OrderElem orderElement : orderElements) {
                        orderVar = (Var) orderElement.getExpr();
                        sqlQuery = sqlQuery + orderVar.getName() + " " +
((orderElement.isAscending()) ? "ASC" : "DESC") + ", ";
                }

                sqlQuery = sqlQuery.substring(0, sqlQuery.length() - 2);
        }

        // writing translated query to log file and console screen
                log.debug(sqlQuery);
                System.out.println("\r\nRunning following query against database.\r\n");
                System.out.println(sqlQuery.replaceFirst("SELECT", "SELECT
").replaceFirst("FROM", "\r\nFROM    ").replaceFirst("WHERE", "\r\nWHERE   ").replaceAll("AND", "\r\nAND
").replaceAll("ORDER BY", "\r\nORDER BY").replaceAll(",", ",\r\n        ").replaceAll("DISTINCT",
"DISTINCT\r\n        "));
        }
        else {
                log.error("Cannot access parsed mapping document or parsed SPARQL query.");
                System.exit(1);
        }

        // returning translated query to main program
        log.info("Query translation completed.");
        return sqlQuery;
    }

    // method to find out how many triple maps contain the predicate map
    private static int noOfMatchesTM (String predicateIn, MappingDocument mappingdocIn){
            int count = 0;

            LinkedList<LogicalTableMapping> tables = mappingdocIn.getLogicalTableMappings();

    for (LogicalTableMapping ltm : tables) {
    ArrayList<PredicateObjectMap> pomaps = ltm.getPredicateObjectMaps();
    for (PredicateObjectMap pomap : pomaps) {

            ArrayList<String> predicates = pomap.getPredicates();

            for (String predicate : predicates) {
                    if (predicateIn.equals(predicate)) {
                            count++;
                    }
            }
    }
    }
            return count;
    }

    // method to return all triple maps that contain a specific predicate map
```

```
	private static List<String> matchingTM (String predicateIn, MappingDocument mappingdocIn){
		List<String> triplemaps = new ArrayList<String>();

		LinkedList<LogicalTableMapping> tables = mappingdocIn.getLogicalTableMappings();

	for (LogicalTableMapping ltm : tables) {
	ArrayList<PredicateObjectMap> pomaps = ltm.getPredicateObjectMaps();
	for (PredicateObjectMap pomap : pomaps) {

		ArrayList<String> predicates = pomap.getPredicates();

		for (String predicate : predicates) {
			if (predicateIn.equals(predicate)) {
				triplemaps.add(ltm.getUri().substring(ltm.getUri().indexOf("#") + 1));
			}
		}
	}
	}
		return triplemaps;
	}

	// method to retrieve the subject template for a specific triple map
	private static LogicalTableMapping getTMSubjectTemplate (String tripleMap, MappingDocument
mappingdocIn){
		LogicalTableMapping subjectTemplate = null;

		LinkedList<LogicalTableMapping> tables = mappingdocIn.getLogicalTableMappings();

	for (LogicalTableMapping ltm : tables) {
	if (ltm.getUri().substring(ltm.getUri().indexOf("#") + 1).equals(tripleMap)) {
		//subjectTemplate = ltm.getSubjectMap().getTemplate().getText();
		subjectTemplate = ltm;
	}
	}
		return subjectTemplate;
	}

	// method for retrieving the column field extracted from the object map for a specific triple map
and predicate map
	private static String getColumnName (String tripleMap, String predicateStr, MappingDocument
mappingDoc) {

		String colName = null;

		LinkedList<LogicalTableMapping> tables = mappingDoc.getLogicalTableMappings();

	for (LogicalTableMapping ltm : tables) {
	tripleMapStr = ltm.getUri().substring(ltm.getUri().indexOf("#") + 1);

	if (tripleMapStr.equals(tripleMap)) {
		ArrayList<PredicateObjectMap> pomaps = ltm.getPredicateObjectMaps();
		for (PredicateObjectMap pomap : pomaps) {

			ArrayList<String> predicates = pomap.getPredicates();

			for (String predicate : predicates) {
				if (predicateStr.equals(predicate)) {
					// first look for column map and if exists use column value
					if (pomap.getObjectColumn() != null) {
						colName = pomap.getObjectColumn();
					}
					// if no column map, must be template map
					else {
						// extract column value from template map
						ArrayList<String> fields =
pomap.getObjectTemplate().getFields();

						// scope defines that there will be only one field, list
has maximum length of one
						for (String field : fields) {
						colName = field;

						}
					}
				}
```

```
                        }
                }
        }
        }
        return colName;
        }

        // method to retrieve the established table identifier and field identifier for a variable in a
triple pattern
        private static String getColumn (String variable) {
                String var = null;

                for (int j = 0; j < projVarTrack.size(); j++) {
                        if (variable.equals(projVarTrack.get(j).get(0))) {
                                var = projVarTrack.get(j).get(1) + "." + projVarTrack.get(j).get(2);
                        }
                }
                return var;
        }

        // method to retrieve the established triple map for a variable in a triple pattern
        private static String getTMforVar (String variable) {
                String tm = null;

                for (int j = 0; j < projVarTrack.size(); j++) {
                        if (variable.equals(projVarTrack.get(j).get(0))) {
                                tm = projVarTrack.get(j).get(1);
                        }
                }
                return tm;
        }

        // method for evaluating compare and regex SPARQL FILTER clauses
        private static List<String> evalFilter (TupleExpr sparqlSelect, List<List<String>> varTrack) {

                List<String> filters = new ArrayList<String>();
                String condition = null;

                // look for compare filters first
        CompareCollector compareVisitor = new CompareCollector();
                sparqlSelect.visitChildren(compareVisitor);
        List<Compare> compares = compareVisitor.getCompare();
        if (!compares.isEmpty()) {
                for (Compare compare : compares) {
                        Var leftVar = null;
                        ValueConstant leftValue = null;
                        if (compare.getLeftArg() instanceof Var) {
                                leftVar = (Var) compare.getLeftArg();
                                condition = getColumn(leftVar.getName());
                        }
                        else if (compare.getLeftArg() instanceof ValueConstant) {
                                leftValue = (ValueConstant) compare.getLeftArg();
                                if (leftValue.getValue().toString().contains("string")) {
                                condition += "'" + leftValue.getValue().stringValue() + "'";
                                }
                                else {
                                condition += leftValue.getValue().stringValue();
                                }
                        }

                        condition += " " + compare.getOperator().getSymbol() + " ";

                        Var rightVar = null;
                        ValueConstant rightValue = null;
                        if (compare.getRightArg() instanceof Var) {
                                rightVar = (Var) compare.getRightArg();
                                condition += getColumn(rightVar.getName());
                        }
                        else if (compare.getRightArg() instanceof ValueConstant) {
                                rightValue = (ValueConstant) compare.getRightArg();
                                if (rightValue.getValue().toString().contains("string")) {
                                condition += "'" + rightValue.getValue().stringValue() + "'";
                                }
                                else {
                                condition += rightValue.getValue().stringValue();
```

```
                        }

                    }
                filters.add(condition);
                }
        }

        RegexCollector regexVisitor = new RegexCollector();
                sparqlSelect.visitChildren(regexVisitor);
        List<Regex> regexpr = regexVisitor.getRegex();
        if (!regexpr.isEmpty()) {
                for (Regex regex : regexpr) {
                        Var leftRegVar = null;
                        ValueConstant leftRegValue = null;
                        if (regex.getLeftArg() instanceof Var) {
                                leftRegVar = (Var) regex.getLeftArg();
                                condition = getColumn(leftRegVar.getName());
                        }
                        else if (regex.getLeftArg() instanceof ValueConstant) {
                                leftRegValue = (ValueConstant) regex.getLeftArg();
                                if (leftRegValue.getValue().toString().contains("string")) {
                                condition += "'" + leftRegValue.getValue().stringValue() + "'";
                                }
                                else {
                                condition += leftRegValue.getValue().stringValue();
                                }
                        }

                        ValueConstant flag = (ValueConstant) regex.getFlagsArg();
                        if (flag.getValue().stringValue().equals("i")) {
                        condition += " ~* ";
                        }
                        else {
                        condition += " ~ ";
                        }

                        Var rightRegVar = null;
                        ValueConstant rightRegValue = null;
                        if (regex.getRightArg() instanceof Var) {
                                rightRegVar = (Var) regex.getRightArg();
                                condition += getColumn(rightRegVar.getName());
                        }
                        else if (regex.getRightArg() instanceof ValueConstant) {
                                rightRegValue = (ValueConstant) regex.getRightArg();
                                if (rightRegValue.getValue().toString().contains("string")) {
                                condition += "'" + rightRegValue.getValue().stringValue() + "'";
                                }
                                else {
                                condition += rightRegValue.getValue().stringValue();
                                }
                        }
                filters.add(condition);
                }
        }

        return filters;
        }
}
```