FILT – Filtering Indexed Lucene Triples

- A SPARQL Filter Query Processing Engine for conventional triplestores –

by Magnus Stuhr

Supervisor: Csaba Veres

Abstract

The Resource Description Framework (RDF) is the W3C recommended standard for data on the semantic web, while the SPARQL Protocol and RDF Query Language (SPARQL) is the query language that retrieves RDF triples by subject, predicate, or object. RDF data often contain valuable information that can only be queried through filter functions. The SPARQL query language for RDF can include filter clauses in order to define specific data criteria, such as full-text searches, numerical filtering, and constraints and relationships between data resources. However, the downside of executing SPARQL filter queries is the frequently slow query execution times. Due to the fact that SPARQL filter queries can retrieve information that non-filter SPARQL queries cannot, decreasing the query execution time of SPARQL filter queries will greatly enhance the efficiency of the SPARQL query language. This thesis presents a SPARQL filter query processing engine for conventional triplestores called FILT (Filtering Indexed Lucene Triples), which is built on top of the Apache Lucene framework for storing and retrieving indexed documents. The objective of FILT was to decrease the query execution time of SPARQL filter queries. This was evaluated by performing a benchmark test of FILT compared to the Joseki triplestore, focusing on two different use-cases; SPARQL regular expression filtering in medical data, and SPARQL numerical/logical filtering of geo-coordinates in geographical locations.

Table of contents

Chapter 1: Introduction

The World Wide Web we know today is built on the architecture of linking documents together as a huge information store, often referred to as “the Web of documents”. These documents are generally expressed with the Hypertext Markup Language (HTML) language in order to tell the computer how to present the information to the users. These documents present readable text that humans can analyze and interpret by putting the information into a specific context. Based on several factors, such as what knowledge domain the user is currently reading about and what the user has searched for, the user can understand the information in the Web documents based on the context being present. However, machines cannot understand the information being displayed to the user, as the information is merely represented by clear text without metadata to tell the machines what the actual text is about. Moreover, the user is on its own when it comes to putting information into a context, exploring relationships and similarity between information, and understanding the information itself. In order to make the machines aiding the users with the tasks of understanding the information better, a new architecture of the World Wide Web has been in the offing. This Web architecture is often referred to as “the Web of Data”, or the “Semantic Web” and tries to deal with the shortcomings of the traditional Web architecture by tagging information with metadata, making data easier to search for and understand for the users. It is built around interlinking data, rather than interlinking text documents. In this text, the Web of data will be referred to as the “Semantic Web”

RDF (Resource Description Framework) is a language for describing things or entities on the World Wide Web (Manola & Miller, 2004). RDF data is structured as connected graphs, and is composed of triples. A triple is a statement consisting of three components: a subject, a predicate and an object. Such a statement can be anything, for instance “Peter has a friend named John”. This could be formally structured as a triple in an RDF graph as this: *Peter hasFriend John*, where Peter would be the subject, hasFriend would be the predicate, and John would be the object. This example is an abstraction of how triples should be structured, as the structure of triples is built around Uniform Resource Identifiers (URIs), literals and blank nodes. Moreover, this means that the subject and predicate, and in many cases the object, are represented by a URI, meaning that they have a unique identifier to represent them. The object of the triple can also be a literal, such as a textual description, a date or an integer. Subjects and objects can also consist of blank nodes – anonymous nodes representing resources where a URI or literal has not been given. RDF data is built on the idea of utilizing unique namespaces/vocabularies for describing data, meaning that every data resource represented by a URI is a part of a unique namespace that identifies what that resource is a part of. For instance, if one would like to specify the latitude of a geo location, one could use the predicate “http://www.w3.org/2003/01/geo/wgs84_pos#lat”, where “http://www.w3.org/2003/01/geo/wgs84_pos#” would be the namespace (the knowledge domain) and “lat” would be the local name of the latitude description within that namespace. Moreover, this means that common knowledge domains and vocabularies can be reused by external data sets, thus making the data more interoperable in terms of sharing, implementing, and interchanging data between different information systems. As opposed to “the Web of documents”, RDF data makes it possible for computers to understand the information they are displaying to the users, meaning that they can help the users put the information into context, inferring and exploring new data relationships, and making searching more accurate and efficient.

Interlinking RDF data is referred to as “Linked Data” (LD) - a term coined by Tim Berners-Lee describing the new generation of the World Wide Web. The idea behind LD is focusing on not just linking *documents* together, but linking *data* together (Berners-Lee, 2006). The purpose of LD is thus giving data meaning to both humans *and* machines by defining unique resources to describe concepts. For instance, if referring to the word “apple” one could specify either the fruit apple or the company “Apple”. Humans can usually make sense of which “apple” the specific text refers to by the given context, but the machines cannot. However, by linking the concept “apple” to a Unique Resource Identifier (URI), a unique resource describing the specific concept, even machines can understand what concepts the text refers to.

Another important aspect that has evolved along with the idea of LD is the Open Data Movement, which focuses on raw data being open and available to everyone. The main purpose behind this movement is that no one should put a barrier around their knowledge-base, but rather share it. Wikipedia is an example of open data with a collective ownership among the community. However, Wikipedia is a website, not a plain data-storage, which makes it hard to query information for re-use. The University of Berlin has made an effort to convert the data from Wikipedia into an open data-storage, named DBpedia (Auer et al., 2007). DBpedia describes the Wikipedia data by applying a local ontology along with numerous external open vocabularies in order to display the enormous amount of data and the relationships between them. There are several other open data sets apart from DBpedia, such as MusicBrainz, Freebase, Linked GeoData, DrugBank, Diseasome and DailyMed, to name a few.

The World Wide Web Consortium (W3C) standard query language for looking up RDF data is the SPARQL Protocol and RDF Query Language, referred to as SPARQL (Prud’hommeaux & Seaborne, 2008). SPARQL makes it possible to retrieve and manipulate RDF data, whether the data is stored in a native RDF store, or expressed as RDF through middleware conversion mechanisms. SPARQL queries are expressed in the same syntax as RDF, namely as triples. To illustrate the syntax of SPARQL queries, this is an example of how a simple SPARQL query can look like:

SELECT ?subject WHERE {?subject ?predicate ?object.}

SPARQL 1.0 became a World Wide Consortium standard for querying RDF data in January, 2008, and has been widely adopted as the leading query language for RDF ever since. The newest version of SPARQL to this day (May, 2012) is SPARQL 1.1 (Prud'hommeaux & Seaborne, 2008). In order to query RDF data through SPARQL, the RDF data itself has to be stored in databases compatible with the SPARQL query language. There are several different database architectures that allow the execution of SPARQL queries – the most common solution being RDF triplestores. Triplestores are databases for storing and retrieving triples. Some triplestores have been built from scratch, while others have been built on existing database solutions, such as relational SQL-based databases. Most triplestores offer a built-in SPARQL endpoint and query interface, making it possible to execute queries and retrieve and manipulate the RDF data stored in the triplestore. SPARQL endpoints are commonly accessed through the HTTP protocol with a query string as a parameter. Most triplestores offer the possibility of retrieving the results of a query in different output formats, such as XML, JSON, CSV or clear text. SPARQL endpoints are also possible to access through programming frameworks, such as Jena for Java (Carroll et al., 2004) and RAP (Oldakowski, et al., 2005) and ARC (Nowack, 2005) for PHP.

As the Web evolves into one enormous database, locating and searching for specific information poses a challenge. RDF data consists of graphs defined by triples, meaning that there are many more relationships and connections between data resources, compared to the traditional Web structure consisting of clear text documents. The RDF data structure offers a more flexible and accurate way of retrieving information, as specific relationships between data resources can be looked up. Moreover, the architecture of the Semantic Web poses a need for another search design opposed to the traditional Web. However, full-text searches will also be important when searching the Semantic Web, as there usually exist a great deal of textual descriptions stored as literals in most RDF data sets. For instance, imagine a triple in an RDF graph describing a fictional book publisher called "Morgan Books" looking like this:

http://library.org/resource/Morgan_Books http://xmlns.com/foaf/0.1/name "Morgan Books"

This triple could easily be looked up by specifying the triple pattern in a query. However, sometimes the users do not know exactly what information are out there, and want to issue more unspecific search terms. For instance, when searching for the book publisher "Morgan Books", searches should also retrieve results from the search input "Morgan". Moreover, full-text searches in RDF data are important, because users often do not know to a full extent what information exists.

SPARQL is a good way of searching for explicit data relationships and occurrences in RDF data sets. SPARQL also offers the possibility of performing full-text searches and filtering terms and phrases through SPARQL filter clauses. These filter clauses enables the filtering of logical expressions and variables expressed in the general SPARQL query. Some of the most frequently used SPARQL clauses are filtering string values, regular expressions, logical expressions and language metadata. In this text, SPARQL queries with filter clauses will be referred to as "SPARQL filter queries", whereas SPARQL queries without filter clauses will be referred to as "general SPARQL queries".

An example of a simple SPARQL filter query looks like this:

SELECT ?subject WHERE {?subject ?predicate ?object. Filter (lang(?object) = 'en').}

The SPARQL filter clause in the example query states that the object variable of the triples found in the data set, represented by the variable "?object", should have a language tag named "en", which is the English language tag. SPARQL filter queries also provide several other possibilities of filtering data in a given data set. Regular expressions can be filtered through SPARQL by applying a "regex" filter clause in the query like this:

SELECT ?s WHERE {?s ?p ?o. Filter regex(?o, "SPARQL regex query")}

This query would return all subjects of triples that had an object value containing the regular expression "SPARQL regex query". Now imagine a data set containing textual descriptions of the treatment of medical conditions. A triple in such a data set could look like this:

http://somenamespace.org/resource/drug01 → http://somenamespace.org/property/canTreat → "Can be used in treatment of headache and nausea"

In order to find drugs related to treating headache and nausea, a SPARQL query looking like this could be executed:

SELECT ?s WHERE {?s < http://somenamespace.org/property/canTreat> ?o. Filter regex(?o, "headache"). Filter regex(?o, "nausea")}

This query would return all the subjects of the triple http://somenamespace.org/resource/drug01 → http://somenamespace.org/property/canTreat → ?o, where ?o contained the regular expressions "headache" and "nausea".

Another example showing the advantage of applying filter clauses in SPARQL queries can be illustrated through a use-case of filtering the numerical values of geographical coordinates, in order to find points of interests on a geographical map. Imagine a data set containing geographical locations, including their latitudes and longitudes, with two triples looking like this:

- *http://somenamespace.org/resource/London →*
  http://www.w3.org/2003/01/geo/wgs84_pos#lat → "51.507221"

- *http://somenamespace.org/resource/London* → *http://www.w3.org/2003/01/geo/wgs84_pos#long* → *"-0.127500"*

Now, imagine a use-case where it is interesting to show geographical points of interest that are nearby London. This could be done by executed a SPARQL query looking like this:

SELECT ?subject WHERE {?subject geo:lat ?lat; geo:long ?long . FILTER ((xsd:float(?lat) - 51.507221 <= 0.30000) && (51.507221 - xsd:float(?lat) <= 0.30000) &&(xsd:float(?long) - - 0.127500 <= 0.30000) && (-0.127500 - xsd:float(?long) <= 0.30000)) }

This query would find all geographical locations within a certain range, in this case 0.30000, of the geographical coordinates of London. This example, along with the SPARQL regex example, show possibilities of finding information that would not be possible through general SPARQL queries without filter clauses.

Unfortunately, SPARQL filter clauses pose a major challenge when it comes to query-execution time. When applying filter clauses in SPARQL queries, the queries have to perform matching of logical expressions or terms and phrases, meaning that the SPARQL queries will execute slower than general SPARQL queries. The execution of SPARQL filter queries will depend greatly on how specific the general SPARQL query is defined, how many filter clauses are being applied to the query, and the size of the data set stored in the data store. If the general SPARQL query is unspecific, meaning that the components of the triples are mainly expressed as variables, even a single filter clause may make the query execute slowly. For instance, the previous geo query specified a constraint on the general query ?subject geo:lat ?lat, where neither the subject nor object were specified. As a result, every subject with a latitude value has to be retrieved and tested against the filter conditions. In the worst case scenario, if the query was ?s ?p ?o Filter() then every single triple of the data set had to be tested against the filter conditions. Moreover, filtering data through SPARQL filter clauses will in many cases lead to slow query execution times, which suggests that there is a huge improvement potential in the query-execution time of SPARQL filter queries.

As SPARQL filter queries can discover data relationships that general SPARQL queries cannot, they play an important role in retrieving RDF data. However, due to the fact that SPARQL filter queries in most cases have a much slower query-execution time than general SPARQL queries; it is easy to shy away from applying filter clauses to the queries. Minack et al. (2008) argue that literals are what connect humans to the Semantic Web, giving meaning and an understanding to all the data that exist on the Web. If literals are taken away from RDF data, the directed graphs that amount to the Web of Data will merely be a set of interconnected nodes that are to a certain extent name- and meaningless. This argument suggests that discovering efficient ways of filtering literals in RDF data will be of great value to the information retrieval aspect of the Semantic Web.

This project aims at discovering new ways of optimizing the query-execution time of SPARQL filter clauses. This has led to an exploration of new ways of storing and retrieving RDF data. Since SPARQL filter queries are mainly based on matching terms, phrases and values in specific data fields, this project will go in the direction of addressing how tools for indexing data can be applied to RDF data, and how such tools can enhance the query-execution time of SPARQL queries. This decision was made due to the fact that indexing tools are made exactly for the reason of quickly looking up expression, terms and phrases in specific data stored in pre-defined index document fields. Based on this, the hypothesis of this project is:

A hybrid database solution using full-text search and numerical/logical filtering for RDF literals, combined with a regular triplestore, is feasible, and will dramatically improve query-execution times.

The specific hypotheses are specified in section 2.2, following the technical background of the project.

Chapter 2: Background

This chapter will present the background of the project. The chapter is divided into two major sections: the technical background, where the technologies and frameworks applied in the project will be described, and the problem area background, where the approach to the problem, research questions and relevant research will be presented.

2.1 Technical background

2.1.1 RDF

The main objective of this project is to address ways of optimizing the query-execution time of SPARQL filter queries. In order to achieve such a thing, it is important to have a thorough understanding of the architecture of RDF data, how such data can be queried through SPARQL, and the underlying technical aspects of SPARQL filter clauses. As mentioned in Chapter 1, RDF is the proposed standard format for exchanging and interlinking data on the Web (Manola & Miller, 2004). RDF is a common framework for describing data that can be exchanged across different applications and systems without loss of meaning. RDF statements are expressed as triples, and consist of a subject, predicate and object. The subject in every triple, representing the entity or concept, must be identified by a URI (Uniform Resource Locator). The same principle applies to the predicate expressed in a triple. The object, however, can be represented either by a URI, literal or a blank node. URIs are unique identifiers that are used to describe unique entities or concepts in order to prevent data ambiguity. Literals are data resources that are not identified as entities, and therefore cannot be expressed as URIs. Examples of such data resources can be a string representation of a title or name, a date, or an integer value. Literals are being expressed as data resources with the attribute rdf:parseType=”Literal”. This way, the data model knows that the data resource is a literal, and can cope with it thereafter. Imagine this statement needed to be expressed as a triple in an RDF data set:

“Morgan Books has the description ‘Morgan Books is a book publisher.’”

The statement could be expressed as a triple like this:

http://library.org/resource/Morgan_Books http://dublincore.org/2010/10/11/dcterms.rdf#description “Morgan Books is a book publisher.”

In this triple statement both the subject *http://library.org/resource/Morgan_Books* and the predicate *http://dublincore.org/2010/10/11/dcterms.rdf#description* are URIs, whereas *“Morgan Books is a book publisher”* is the literal. It is a non-unique data resource, thus cannot be expressed as a URI.

Blank nodes are often referred to as anonymous nodes and are used if the subject of an RDF sub-graph is unknown, or if the sub-graph simply does not need to be accessed outside its superior graph. For instance, a subject in a statement is unknown if the data set expresses that “The book-publisher “Morgan Books” has published a book in the year of 1990”. This statement does not assign an identifier to the specific book that has been published in 1990 - it simply states that an undefined book has been published in that year. The book-entity itself is unknown. This could be expressed as triples like this:

http://library.org/resource/Morgan_Books http://library.org/property/hasPublished :_b1

:_b1 http://library.org/property/publicationYear “1990”^^xsd:date

Triples must be expressed in an RDF compatible format, meaning that the triple syntax can be parsed as RDF data. There are three different standard formats for expressing RDF triples: RDF/XML, N-Triples and Turtle. The RDF/XML (Beckett, 2004) syntax is, as the name suggests, an XML notation of RDF. This means that RDF triples are expressed in XML syntax. An example of an expressed RDF/XML data entity with the URI “http://www.library.org/resource/Morgan_Books” looks like this:

```
<?xml version="1.0"?>
< xmlns:libraryProperty="http://www.library.org/property/">

<rdf:Description rdf:about="http://www.library.org/resource/Morgan_Books">
<libraryProperty:foundedIn>December 12, 1985</libraryProperty:foundedIn>
<libraryProperty:locatedIn>The North Pole</libraryProperty:locatedIn>
</rdf:Description>

</rdf:RDF>
```

The RDF/XML syntax uses abbreviations when expressing URIs within a specific namespace. The adoption of XML namespaces is a W3C recommendation (Bray et al., 2006) and is defined by a URI referring to a domain of concepts, terms or entities. An example of a namespace is “http://www.library.org/property/”, which was defined in the RDF/XML description of the resource “http://www.library.org/resource/Morgan_Books”, just given as an example. Namespaces are declared by mapping abbreviations, called prefixes, to the full namespace URI. When expressing the data set it is then possible to refer to the prefixes mapped to the namespace URIs, instead of having to state the full URI every time the namespace is referred to. This is demonstrated in the RDF/XML example through the predicates “libraryProperty:foundedIn” and “libraryProperty:locatedIn”, where the namespace URI has been mapped to a prefix in the beginning of the document by stating:

<rdf:RDF xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
xmlns:libraryProperty="http://www.library.org/property/">

By mapping prefixes to namespaces in the beginning of the document, the full URI “http://www.library.org/property/” can be referred to as the prefix “libraryProperty” throughout the data set, making the data more readable for humans, take up a lesser amount of disk-space and less time consuming to manually express (by having to write less characters). The Turtle syntax (Beckett & Berners-Lee, 2011) is similar to the RDF/XML in terms of defining namespaces and expressing the actual triples, but the syntax still differs to a great extent. The RDF graph describing the entity “http://www.library.org/resource/Morgan_Books” expressed in the RDF/XML example can be expressed in the Turtle syntax like this:

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix libraryProperty: < http://www.library.org/property/>
@prefix libraryResource < http://www.library.org/resource/>

```
libraryResource:Morgan_Books
        libraryProperty:foundedIn “December 12, 1985” ;
        libraryProperty:locatedIn “The North Pole” .
```

Just as in the RDF/XML syntax, the Turtle syntax also defines namespaces and their prefixes in the beginning of the document. There are also similarities between the RDF/XML and Turtle syntaxes when it comes to expressing triples, as both the syntaxes group triples together by only stating the subject URI once. The predicates and objects of the subject URI can then be expressed, divided by a semicolon. When the triples referring to a given subject URI have been expressed, a dot must be entered in order to state that the subject URI and its related predicates and objects have all been expressed. The N-Triples format (Grant & Beckett, 2004) differs from the RDF/XML and Turtle syntaxes in this aspect by simply expressing every triple in the RDF data set as a separate line. Also, the N-Triples syntax does not apply prefixes, thus referring to the full namespace URI whenever referring to a URI. Moreover, the syntax is purely based on expressing every triple of a given data set explicitly. The N-Triples syntax of the same data entity expressed in the RDF/XML and Turtle syntaxes examples looks like this:

<http://www.library.org/resource/Morgan_Books> <http://www.library.org/property/foundedIn>
“December 12, 1985”.

<http://www.library.org/resource/Morgan_Books> <http://www.library.org/property/locatedIn> "The North Pole".

The N-Triples syntax is usually adopted for large RDF dumps, because the syntax offers the possibility of being read line by line, due to the fact that every line expresses a full statement and are not dependent on other lines to make sense. This makes the N-Triples syntax more manageable for machines, meaning that the data sets do not have to be loaded as entire data sets into a system, thus coping with issues such as lack of memory.

The triple statements of an RDF data set are grouped together as graphs of nodes and arcs. Nodes in an RDF graph represent the subject and the object of a triple, whereas arcs represent the predicates of every triple. The triples are grouped together based on their subject URI, meaning that statements about every data entity or concept described in the data model are grouped together as sub-graphs of the default RDF graph. See Figure 2.1 for a basic RDF graph model based on the example of the data entity *http://www.library.org/resource/Morgan_Books.*

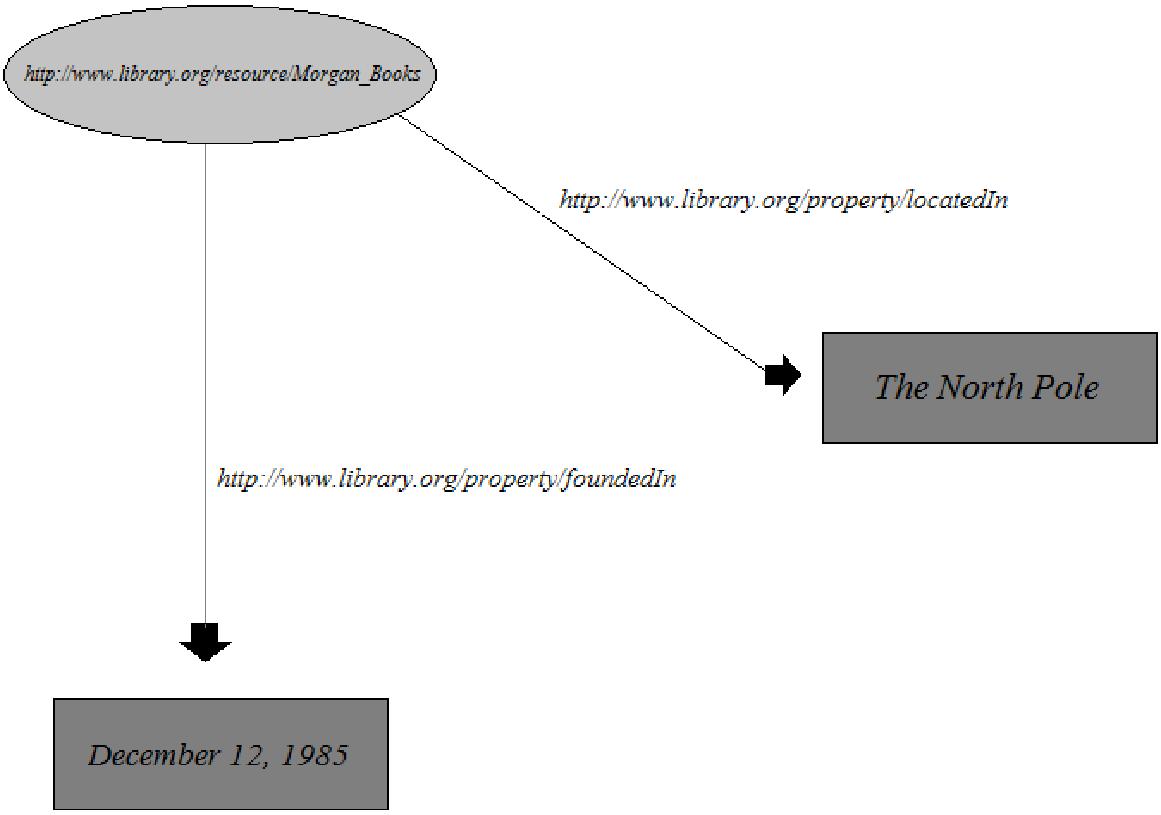

http://www.library.org/resource/Morgan_Books
http://www.library.org/property/locatedIn
The North Pole
http://www.library.org/property/foundedIn
December 12, 1985

Figure 2.1: The RDF graph of the data entity http://www.library.org/resource/Morgan_Books

Moreover, RDF data convey information as graphs consisting of triples. In order to query these data they have to be stored in an RDF compatible database. The most commonly used solution for storing

RDF data and making them accessible through querying is a triplestore. Triplestores are specifically designed to store RDF data, and most triplestore solutions provide a reasoning engine for inferring new triples based on existing ones, and a data access API; most often a SPARQL endpoint. The next section will cover the fundamental aspects of the SPARQL query language and how SPARQL queries are executed over RDF data sets.

2.1.2 SPARQL

As mentioned in Chapter 1, SPARQL is a World Wide Consortium (WC3) standard for querying RDF data. SPARQL queries are executed over an RDF data set consisting of one default graph, representing a collection of sub-graphs (Prud'hommeaux & Seaborne, 2008). SPARQL queries can match graph-patterns in a data set by expressing such graph-patterns in the queries. These graph-patterns are sets of triple patterns and are matched against triple patterns in the data set.

There are four different forms of SPARQL queries (Prud'hommeaux & Seaborne, 2008):

- SELECT
- DESCRIBE
- ASK
- CONSTRUCT

The SELECT form makes it possible to define what data resources (expressed as variables) that should be returned from the query, based on what data relationships and constraints defined in the query itself. An example of a simple SPARQL SELECT query looks like this:

SELECT ?s WHERE {?s ?p ?o}

This query will return all results matching the ?s variable, in this case the subject of every triple in the data set, as all the components of the triple defined in the query are represented as variables. A more specific SPARQL query example looks like this:

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX libraryProperty: <http://www.library.org/property/>
PREFIX libraryResource <http://www.library.org/resource/>

SELECT ?locatedIn WHERE {libraryResource:Morgan_Books libraryProperty:locatedIn ?locatedIn}

This query example would find the variable "?locatedIn", which is the object of the triple *"libraryResource:Morgan_Books libraryProperty:locatedIn ?locatedIn"* presented in the query graph-pattern. This query would return the location of where the library entity "libraryResource:Morgan_Books" is located.

If the character “*” is expressed instead of any variable, all the variables defined in the query will be retrieved as output to the query. The DESCRIBE form differs from the SELECT form in terms of describing the entire RDF graph of the variable defined in the DESCRIBE solution sequence, instead of simply retrieving specific variables, such as the SELECT solution sequence offers. The syntax of a simple DESCRIBE query with the purpose of describing the entire RDF graph of one specific data entity looks like this:

DESCRIBE <URI>

For instance, a DESCRIBE query can be expressed to retrieve the entire RDF graph of the http://www.library.org/resource/Morgan_Books entity like this:

PREFIX libraryResource <http://www.library.org/resource/>

DESCRIBE libraryResource:Morgan_Books

DESCRIBE queries can also describe entities that are constrained by specific relationships defined in the query itself. An example of this is the following query:

DESCRIBE ?s WHERE {?s ?p ?o}

This query would return the RDF graph of every subject in a given data set, as all the components of the triple defined in the query are represented as variables. The RDF data returned by a SPARQL query is not predetermined by the query itself, as the query client would need to know the structure of the RDF in the data store. Instead, the structure of the data returned is defined by the SPARQL query processor. Moreover, the query pattern defined in the SPARQL query is merely applied to create a result set, and the format of the data description itself depends on the SPARQL query service.

A more specific example of a DESCRIBE query looks like this:

PREFIX libraryProperty: <http://www.library.org/property/>

DESCRIBE ?s WHERE {?s libraryProperty:locatedIn “The North Pole”}

This would describe the RDF graph of the “?s” variable, which in this case is the subject of the triple *“?s libraryProperty:locatedIn “The North Pole””.*

The CONSTRUCT query form allows for constructing customized RDF graphs based on the data represented in the data set. The result set is returned as a single RDF graph specified by a pre-determined graph template. This RDF graph is constructed by taking each of the query solutions in the solution sequence and combining the triples returned into one RDF graph. An example of a SPARQL CONSTRUCT query looks like this:

CONSTRUCT {?s ?p ?o} WHERE {?s ?p ?o}

This query would construct an RDF graph out of the entire data set, as all of the values of the triple components in the graph-pattern expressed in the query are variables. An example of a more specific CONSTRUCT query looks like this:

PREFIX libraryProperty: <http://www.library.org/property/>

CONSTRUCT {?s libraryProperty:locatedIn ?locatedIn} WHERE {?s libraryProperty:locatedIn ?locatedIn; libraryProperty:foundedIn "December 12, 1985"}

This CONSTRUCT query would construct a new RDF graph consisting of the triple *?s libraryProperty:locatedIn ?locatedIn* for every data entity that corresponded to the graph pattern defined in the WHERE clause.

The ASK SPARQL query form can be used to check if the graph-pattern expressed in the query has a solution in the data set. No data is returned about the actual query solutions, rather the query simply returns either of the two Boolean values "true" or "false", based on if the solutions exist or not. An example of a general ASK SPARQL query looks like this:

ASK {?s ?p ?o}

This query would simply ask if any triple pattern exists in the data set. For any RDF data set, this query would return true. A more specific ASK query, based on the library example that was used in section 2.1 looks like this:

PREFIX libraryResource: <http://www.library.org/resource/Morgan_Books>
PREFIX libraryProperty: <http://www.library.org/property/locatedIn>

ASK {libraryResource:Morgan_Books libraryProperty:locatedIn ?o }

Based on the RDF graph description of the *http://www.library.org/resource/Morgan_Books* entity described in section 2.1, this query would return true, as the RDF graph contains the triple pattern expressed in the query.

2.1.2.1 SPARQL filter clauses

SPARQL filter clauses restrict the query solutions of a given graph pattern match corresponding to a specified constriction (Prud'hommeaux & Seaborne, 2008). Filter clauses exclude any solutions that are not bound by a specific constraint, meaning solutions that has a Boolean value of false or produce an error. There are a large number of filter functions through SPARQL queries, and this section will not cover them all. An elaboration on every SPARQL filter clause can be found in the SPARQL 1.1

W3C description (Prud'hommeaux & Seaborne, 2008). This text will present some of the most commonly used SPARQL filter clauses that are highly relevant for this project. These filter clauses are:

- regex
- str
- lang
- isIRI
- isLiteral
- datatype
- RDFterm-equal
- logical expressions

The "regex" filter clause uses the XPath fn:matches function to match text against a regular expression pattern (Prud'hommeaux & Seaborne, 2008). The regular expression syntax is presented by Malhotra et al. (2010). An example of a "regex" filter clause is illustrated in the following SPARQL query:

*SELECT * WHERE {?s ?p ?o. Filter regex(?o, "SPARQL regex query")}*

The "regex" filter clause in this query will filter through any object component of a triple and match the value "SPARQL regex query" through the XPath fn:matches function to match the text input against a regular expression pattern. Moreover, this means that if an object literal in the dataset was to contain the string value "This is a SPARQL regex query for explaining the regex filter clause", the "regex" filter clause would return true, as the XPath fn:matches function simply matches the regex value to appear in the string value. In other words, every triple that included the regular expression "SPARQL regex query" in the object literal would be returned.

The "str" filter clause returns the lexical form of a literal and the code point representation of an IRI (Prud'hommeaux & Seaborne, 2008). An example of a "str" filter clause is illustrated in the following SPARQL query:

*SELECT * WHERE {?s ?p ?o. Filter (str(?o) = "SPARQL str query")}*

This "str" filter clause in this query will match every object of every triple in the data set to have the exact value "SPARQL str query". Opposed to the "regex" filter clause, which matches expressions to appear in a string value, the "str" filter clause matches the whole string value of a triple-component to match the entire filter value.

The “lang” filter clause returns the language tag of a literal. It returns an empty string if the literal has no language tag (Prud’hommeaux & Seaborne, 2008). An example of a “lang” filter clause is demonstrated in the query:

*SELECT * WHERE {?s ?p ?o. Filter(lang(?o) = “en”)}*

The “lang” filter clause in this query will match every object of a triple that has an English language tag. The filter values of the “lang” filter clause, in this case “en”, is based on the ISO 639 two-letter language codes (The US Library of Congress, 2010) and is expressed in the data set by applying the “xml:lang” annotation (Biron & Malhotra, 2004).

The “isIRI” filter clause returns true if a term is an Internationalized Resource Identifier (IRI), and false otherwise (Prud’hommeaux & Seaborne, 2008). IRIs are generalizations of URIs and contain a sequence of characters from the Universal Character Set, Unicode/ISO 10646 (Duerst & Suignard, 2005). The “isLiteral” filter clause returns true if a term is a literal and false otherwise (Prud’hommeaux & Seaborne, 2008). An example of a SPARQL query containing both the “isIRI” and the “isLiteral” filter clauses looks like this:

*SELECT * WHERE {?s ?p ?o. Filter(isIRI(?s)). Filter (isLiteral(?o))}*

This query will match the variable ?s to be an IRI, and the variable ?o to be a literal. The query will only return true if both filter clauses return true, meaning that ?s must be an IRI and ?o must be a literal.

The “datatype” filter clause returns the data type IRI of a literal (Prud’hommeaux & Seaborne, 2008). The filter clause operates based on these criteria:

- If the literal is a typed literal, return the data type IRI.
- If the literal is a simple literal, return xsd:string
- If the literal is literal with a language tag, return rdf:langString

An example of a SPARQL query containing a “datatype” filter clause looks like this:

*SELECT * WHERE {?s ?p ?o. Filter(datatype(?o) = xsd:double)}*

The “datatype” filter clause in this query matches the value of the variable ?o to be of the data type “double”, defined in the XML schema (Biron & Malhotra, 2004). This query will only return true if the value of the variable ?o, in the object of any triple in the data set, is of the data type “double”.

It is also possible to filter logical expressions through SPARQL. “RDFterm-equal” is a filter clause that operates with logical expressions, in this case processing the equality, or lack thereof, between two RDF terms. The "RDFterm-equal" filter clause returns true if term1 and term2 are the same RDF terms (Prud'hommeaux & Seaborne, 2008). term1 and term2 are the same if any of the following are true:

- term1 and term2 are equivalent IRIs as defined in http://www.w3.org/TR/rdf-concepts/#section-Graph-URIref
- term1 and term2 are equivalent literals as defined in http://www.w3.org/TR/rdf-concepts/#section-Literal-Equality
- term1 and term2 are the same blank node as described in http://www.w3.org/TR/rdf-concepts/#section-blank-nodes

The "logical expression" filter clauses do not have a filter clause identifier attached to them, such as the previous filter clauses described in this section. For instance, the "regex" filter clause is identified by the name of the filter clause, such as "Filter regex(?o, "SPARQL")", whereas "logical expression" filter clauses have no such identifier. This text will refer to all filter clauses with no filter clause identifier attached to them as "logical expression" filter clauses. Based on this, two different examples of SPARQL queries implementing the "logical expression" filter clause defined in this text look like this:

1. *SELECT * WHERE {?s ?p ?o. Filter(?o != 50)}*
2. *SELECT * WHERE {?s ?p ?o. Filter(?o >= 75)}*

The filter clause in the first query matches every triple where the variable ?o is not equal to the value 50. This query only returns true where any triple matching the triple pattern ?s ?p ?o does not have a value of 50 in the ?o variable. The filter clause in the second query matches every triple where ?o equals or have a higher value than 75. The two example queries can also be merged into one query like this:

*SELECT * WHERE {?s ?p ?o. Filter(?o != 50 && ?o >= 75)}*

In this query the two filter clauses of the first and second query has been merged by applying the Boolean operator AND ("&&"). This means that both filter expression must be true in order for the entire query to return true.

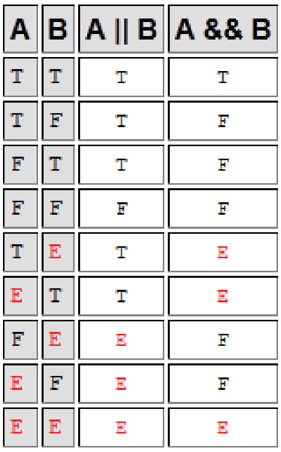
2.1.2.2 SPARQL FILTER Evaluation

SPARQL provides a subset of the functions and operators defined by the XQuery Operator Mapping (Prud'hommeaux & Seaborne, 2008). Boag et al. (2010) define the calling of XPath functions. The execution of functions through SPARQL is defined as "SPARQL Filter Evaluation". There are certain rules that hold the differences in how functions execute in XQuery opposed to SPARQL (Prud'hommeaux & Seaborne, 2008). The rules are as following:

- SPARQL functions differ from XPath/XQuery functions in terms of SPARQL functions not processing node sequences. SPARQL functions presume that any argument is a sequence of a single node.
- If a function is called with an argument of the wrong type a type error will occur. Type errors are described in the XQuery 1.0 specification (Boag et al., 2010).
- All functions and operators, except the “bound”, “coalesce”, “not exists” and “exists” handle RDF Terms and will generate a type error if any arguments are not bound.
- Any expression where an error is present will generate the given error, apart from logical-or (||) and logical-and (&&) expressions.
- A logical-and function that has an error in one branch will return an error if the other branch is true, and false if the one or more of the other branch is false.
- A logical-or function that has an error in one branch will return an error the other branch is true, and false if the other branch is false.
- A logical-or or logical-and function that has an error on both branches will generate one of the two previously described errors.

The logical-and and logical-or truth conditions for filtering variables by using the logical operators “AND” and “OR” is shown in Table 1.1. This table is taken from the SPARQL 1.1 specification (Prud’hommeaux & Seaborne, 2008). The table operates with “T” for true, “F” for false, and “E” for error.

Table 1.1: The logical-and and logical-or truth conditions



A	B	A || B	A && B
T	T	T	T
T	F	T	F
F	T	T	F
F	F	F	F
T	E	T	E
E	T	T	E
F	E	E	F
E	F	E	F
E	E	E	E

When calling functions on more than one argument, SPARQL follows this syntax for handling the functions (Prud’hommeaux & Seaborne, 2008):

- Argument values are generated based on the argument expressions that are evaluated. The order of which the arguments are evaluated in is undefined.
- Numerical values expressed as arguments are arranged to fit the expected types for that specific function or operator
- The given function or operator is called on the argument values

If any of these steps fail to execute, type errors are generated accordingly.

The next two sections in the thesis will highlight the Apache Lucene and the Apache Jena for the Java programming language. These two frameworks have an important role in the technical solution of this project.

2.1.3 Apache Lucene

Apache Lucene is a free open-source high-performance information retrieval engine written in the Java Programming language. It offers full-featured text search, based on indexing mechanisms (Apache Lucene, 2011). Lucene is a vital part of storing and querying data in FILT, a database solution developed in this project, which will be presented in detail later in the thesis. This section will describe the foundational technical aspects of the Apache Lucene framework.

2.1.3.1 Indexing documents with Lucene

A Lucene index contains a set of documents which again contains one or more fields. These fields can be stored as text or numerical values, and can either be analyzed or not analyzed by the Lucene library, which will later affect how the given information can be retrieved. Moreover, a Lucene Document Field is a separated part of a document which can be indexed so that terms in the field can be used to retrieve the document through Lucene queries. To illustrate this, imagine a Lucene document describing “Football”, containing a field named “title” and a field named “description”. This document would look like this:

```
Document {
    Field {
            name: title
            value: Football
    }
    Field {
            name: description
```

```
        value: Football is a sport.
    }
}
```

The document groups the document fields together, meaning that the field named “title” will be seen in the context of the field named “description”. By adding new documents, new data instances are created, meaning that the same field names, in this case “title” and “description”, can be used to describe other data instances. For instance, a new document describing the sport “basketball” could be created by specifying the same field names as were used in the “football” example like this:

```
Document {
    Field {
        name: title
        value: Basketball
    }
    Field {
        name: description
        value: Basketball is a sport.
    }
}
```

The fields in this example would not overwrite the fields in the “football” example, as the “football” data instance is located in another document, thus being treated as separate data instance. Based on the document structure presented in the recent examples, Lucene queries can be executed in order to find the title and description of a document. For instance, if one wanted to find the document containing the field named “title” with the value of “Football”, a query looking like this could be executed:

title:Football

This query would return the document containing the information about Football, as presented in an earlier example. Further, the user could call methods on the document being returned in order to retrieve specific fields from the documents, such as the “description” field. Lucene queries will be explained in detail in section

2.1.3.2 Querying documents with Lucene.

An analyzed index field is divided into several sub-terms based on the text input value, meaning that the information can be retrieved by specifying one or more terms that occur in the text, instead of having to provide the full text as a search input in order for the index to locate the information. Analyzing fields also makes it easier to retrieve information based on closely related search-terms, which are not necessarily matching the exact same terms in the text that was indexed. This is made possible by running the input value through a field analyzer. There are several diverse built-in analyzers in the Lucene library that can be used to analyze the indexed information, each of them analyzing text differently. Analyzed fields are advantageous for indexing structured text, such as content descriptions, making it easy to perform full-text search based on frequently used terms in a text or terms that are closely related. It is fully possible to write one's own analyzers and also use different analyzers on each field in the index. Lucene offers a way of analyzing fields differently through the "PerFieldAnalyzerWrapper" class, which lets one associate a different analyzer with different fields (Lucene API, 2012). Table 2.1 lists the names and short descriptions of the most commonly used analyzers for analyzing index fields in Lucene.

Table 2.1: Different index analyzers in Lucene (Apache Lucene API, 2011)

Name	Short description
StandardAnalyzer	Filters StandardTokenizer with StandardFilter, LowerCaseFilter and StopFilter, using a list of English stop words
SimpleAnalyzer	An Analyzer that filters LetterTokenizer with LowerCaseFilter
StopAnalyzer	Filters LetterTokenizer with LowerCaseFilter and StopFilter
KeywordAnalyzer	"Tokenizes" the entire stream as a single token. This is useful for data like zip codes, ids, and some product names.
WhitespaceAnalyzer	An Analyzer that uses WhitespaceTokenizer
LimitTokenCountAnalyzer	This Analyzer limits the number of tokens while indexing. It is a replacement for the maximum field length setting inside IndexWriter

Non-analyzed fields are not being interpreted and manipulated by the Lucene library, and will have the same state as the input specified into the index field. Non-analyzed fields are particularly purposeful for indexing database keys, IDs, telephone numbers and other information that are meant to be looked up by giving the complete data value as a search input. Table 2.2 shows the different possibilities when it comes to determining how an index field should be analyzed or not, based on the official Lucene documentation (Apache Lucene API, 2011).

Table 2.2: Different index field analyzer attributes (Apache Lucene API, 2011)

Index attribute	Short description
ANALYZED	Index the tokens produced by running the field's value through an Analyzer.
ANALYZED_NO_NORMS	Expert: Index the tokens produced by running the field's value through an Analyzer, and also separately disable the storing of norms.
NO	Do not index the field value.
NOT_ANALYZED	Index the field's value without using an Analyzer, so it can be searched.
NOT_ANALYZED_NO_NORMS	Expert: Index the field's value without an Analyzer, and also disable the indexing of norms.

It is also possible to choose whether or not an index field should be stored. This attribute determines if a value of a given index field can be retrieved from the index once stored (see Table 2.3). If a field is stored in the index, the value of that field can be retrieved through Lucene as output. On the other hand, a field that is not stored is only possible to query, and not possible to retrieve as output. The storing attribute has to be applied during the indexing process, and cannot be changed at a later stage without having to perform the indexing process all over again. The index consumes more disk-space if a document-field value is stored, opposed to the value not being stored.

Table 2.3: Different index field store attributes in Lucene (Apache Lucene API, 2011)

Store attribute	Store description
YES	Stores the specific document field as available output
NO	Do not store the specific document field as available output

2.1.3.2 Querying documents with Lucene

In addition to offering ways of storing data and information as indices, the Apache Lucene framework also provides an extensive library for querying such indices. There exist a wide range of different querying possibilities depending on the data that should be looked up. This section will shed light on the basic querying principles in Lucene, as well as presenting some of the most commonly used queries and their use.

First of all, the results output of queries executed through a Lucene index depend on how the index is constructed in terms of what analyzers have been applied to the document-fields. However, the query execution itself will be the same regardless of how the index is structured. A Lucene query is broken up into terms and operators. Terms can either be composed as one single term, such as “Football”, or as phrases, such as “Football player”. For example, in order to find documents with the title “Football”, one could specify a term query looking like this:

title:”Football”

This query looks for the value “Football” in documents containing a field named “title”. If one rather wanted to search for the title containing the phrase “Football player”, one could specify a phrase query like this:

title:”Football player”

Finally, multiple terms can be merged together through Boolean operators in order to form more intricate queries (Apache Lucene Query Parser Syntax, 2012). For instance, if one wanted to find documents with the title “Football” or “Football player”, one could compose a query looking like this:

title:”Football” OR title:”Football player”

Lucene also provides the possibility of specifying a range between different terms to be fulfilled. For instance, if one has an index consisting of data about persons, where one of the document-fields contain the age of these persons, one could find all persons with the age between 20-25 by constructing a range query like this:

age:[20 TO 25]

This query would find all persons with the age of 20, 21, 22, 23, 24 or 25, as the square brackets around the term range indicate that the minimum and maximum value should be inclusive in the query. In order to exclude the minimum and maximum range of the search, the square brackets should be replaced with curly brackets like this:

age:{20 TO 25}

Range queries can also be applied to other data types than integers, such other number formats, strings and dates.

The examples above construct queries as query-strings that can be provided as input to the main QueryParser class in Lucene. This is a good way of translating natural language queries into formally structured queries that can run through Lucene, but this querying method has its restrictions. For instance, the QueryParser class will remove all special characters from the query-string, meaning that what the users can provide as search input is to a great extent limited. Moreover, this way of querying can only take simple natural language query-strings and execute them through the index. However, Lucene offers a wide range of different query classes for handling more complex querying. Some of the most commonly used queries are presented in Table 2.4. For instance, if one specifies a single term or a phrase that should match a value in a document-field, a TermQuery or PhraseQuery will be most suited. However, if a term or a phrase includes regular expressions, then a RegexQuery will be the best alternative. The TermRangeQuery is suited for finding terms within a range, for instance finding all persons with the name between “Alan” to “Donald”, whereas the NumericRangeQuery is appropriate for filtering numeric values with the same principle. Further, queries can be combined into more complex queries through the BooleanQuery class. This class provides the possibility of merging queries and adding Boolean operators between them.

Table 2.4: A selection of the built-in query classes in Lucene (Apache Lucene API, 2011)

Query class	Short description
TermQuery	A Query that matches documents containing a term.
PhraseQuery	A Query that matches documents containing a particular sequence of terms.
RegexQuery	Implements the regular expression term search query.
TermRangeQuery	A Query that matches documents within a range of terms.
NumericRangeQuery	A Query that matches numeric values within a specified range. To use this, you must first index the numeric values using NumericField
BooleanQuery	A Query that matches documents matching Boolean combinations of other queries

The BooleanQuery class lets one combine queries with three different Boolean operators. In Lucene, these operators are constructed through the Occur class, and they can have the values "MUST", "MUST_NOT" and "SHOULD". The "MUST" operator defines that a query must appear in the document in order for the query to be true, the "MUST_NOT" operator defines that a query must not appear in the document in order for the query to be true, and the "SHOULD" operator defines that a query should, but does not have to appear in the document. However, if the BooleanQuery only consists of one or more queries combined with the "SHOULD" operator, one of the queries must be true in order for the BooleanQuery to return any results. Table 2.5 shows the different Boolean operators in the Occur class.

Table 2.5: The Boolean operators in the Occur class (Apache Lucene API, 2011)

Boolean operator	Short description
MUST	Use this operator for clauses that *must* appear in the matching documents.
MUST_NOT	Use this operator for clauses that *must not* appear in the matching documents.
SHOULD	Use this operator for clauses that *should* appear in the matching documents.

2.1.4 Apache Jena

Jena is a Java-framework for building semantic web applications. It was originally developed by HP labs, located in Bristol, UK, in 2000. In 2009, HP decided not to continue working on Jena, though still supporting the entire project. The developers successfully managed to transfer the project to the Apache Software Foundation in November, 2010, and ever since the project has been a part of the Apache license. The latest Jena release is in this moment in time is 2.7.0, and was released in December, 2011.

Jena makes it possible to read, write and manipulate semantic data models, as well as including inference- and SPARQL-engines (Carroll et al., 2004). To go into more detail, Jena includes an API for writing, reading and manipulating RDF data in the RDF/XML, N-Triples and Turtle formats, an ontology API to interact with OWL and RDFS ontologies, reasoning with RDF data sources based on a built-in inference engine, storing RDF data in internal memory and on disk, SPARQL query engine compatible with the latest SPARQL version, and servers that allow RDF data to be published across different applications using diverse protocols, thereby SPARQL (Carroll et al., 2004).

Jena is first and foremost used in FILT for building a local RDF graph based on the relevant triples retrieved from the index, based on the filter clauses in the SPARQL queries that are executed. Jena is also used to run the general SPARQL queries that are stripped of filter clauses over the local RDF model generated from the index. This is described in detail in section 3.3.3.

2.2 Problem area background

2.2.1 Approach to the problem

As presented in both Chapter 1 and previous sections in this chapter, SPARQL filter queries provide multiple possibilities of finding information that could not be found through general SPARQL queries without filter queries. However, the downside of SPARQL filter queries is that these queries generally execute slowly. Instead of simply matching a graph-pattern, which is the case in general SPARQL queries, SPARQL filter queries have to filter through a wide variety of data values stored in the triples. This will naturally lead to slower query execution times opposed to general SPARQL queries. Based on this, this project aims at discovering techniques and principles for optimizing the query-execution times of SPARQL filter queries, and building a prototype solution called FILT to show that the query-execution time of SPARQL queries can be decreased noticeably by implementing the Apache Lucene framework for performing full-text searches and filtering logical/numerical expressions.

2.2.2 Use-cases

To illustrate the problem of SPARQL filter queries executing slowly, there will be presented two use-cases that will lay the foundation for the implementation and focus areas of FILT. The two use-cases aim at illustrating two major aspects in terms of executing SPARQL filter queries, namely filtering regular expressions and filtering numerical values. The first use-case involves finding medical data based on regular expression filtering, and the second use-case includes finding geo-locations by filtering the numerical values that constitute to their geo-coordinates, in this case latitude and longitude.

2.2.2.1 Finding information about drugs based on regular expressions

DrugBank is a data set consisting approximately 6711 (number retrieved from the homepage http://www.drugbank.ca/) FDA-approved (the U.S. Food and Drug Administration) small molecule and biotech drugs (Wishart et al., 2006), and contains detailed information about drugs, including chemical, pharmacological and pharmaceutical data. It also includes widespread drug data such as structure and sequence, as well as drug interactions, drug targets, enzymes and references to research publications. The University of Berlin has made a successful effort in publishing the DrugBank data set as Linked Data on the Web. The data set can be accessed at http://www4.wiwiss.fu-berlin.de/drugbank/. The published data set consists of approximately 765,936 triples and 59,661 RDF links to other Linked Data sources such as the datasets DBpedia, LinkedCT, DailyMed, Diseasome, Bio2RDF's CAS, ChEBI, GeneID, HGNC, IUPAC, KEGG Compound, KEGG Drug, PDB, PFAM and SwissProt.

The DrugBank data set supports the principles of evidence-based medicine in terms of referencing data to scientific publications. Evidence-based medicine refers to the method of finding, evaluating and applying concurrent empirical evidence as the basis for clinical decision-making (Rosenberg & Donald, 1995). For a long time there has been a difference between empirical proof and clinical practice, which may lead to expensive, ineffective or harmful decision making by doctors. Thus, evidence-based medicine include asking questions, finding and assessing data, and using research evidence as a basis for clinical practice (Rosenberg & Donald, 1995).

Evidence-based medicine consists of four steps (Rosenberg & Donald, 1995):

1. "Formulate a clear clinical question from a patient's problem
2. Search the literature for relevant clinical articles
3. Evaluate (critically appraise) the evidence for its validity and usefulness
4. Implement useful findings in clinical practice"

The linked data sets are a good basis for gathering facts according to the evidence based medicine, as it provides the possibility of querying explicitly defined data resources and relationships between them. In this context, triples will be referred to as "explicitly stated data relations". There also exist great deals of useful data in literals, such as mere textual descriptions of data entities. Literals can in some cases provide a thorough understanding of a given data entity. In this context, literals of the data type http://www.w3.org/2001/XMLSchema#string will be referred to as "implicitly stated data relations".

For instance, a textual description of a data resource usually implicitly states data relations between the given data entity and other data resources. For instance, have a look at the DrugBank drug entity of the drug "Diazepam", mainly used for treating anxiety disorders: http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00829. The entity of Diazepam contains a great deal of "implicitly stated data" described in literals, such as the triple:

http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00829 → http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/indication → ?object

The ?object variable in this triple currently has the following value:

"Used in the treatment of severe anxiety disorders, as a hypnotic in the short-term management of insomnia, as a sedative and premedicant, as an anticonvulsant, and in the management of alcohol withdrawal syndrome."

There is a great deal of useful information in this literal, as it describes what the drug is used for. However, looking up this data is tricky, as the data relations are not fragmented into separate triples referring to specific data entities, or terms. Imagine a use-case where a doctor wanted to look up drugs that are used in the treatment of severe anxiety disorders. It would be easier to look up what the drug is used for treating by dividing the literal into several triples, such as this:

- *http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00829 → http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/indication → "Severe anxiety disorders"*
- *http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00829 → http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/indication → "Insomnia"*
- *http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00829 → http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/indication → "Alcohol withdrawal syndrome"*

However, in the DrugBank data set the literals describing the use-case scenarios drugs are described in textual descriptions, consisting of full sentences. These data relations can be retrieved through real-time queries, but due to the fact that implicitly stated data relations are not explicitly stated as triples, looking up these data relations can be tricky. The implicit data relations can be found by filtering literals or URIs using regular expressions in SPARQL queries. Regular expression filtering can be executed through the "regex" filter clause described in section 3.3.1.1. Filtering regular expressions is useful for looking up terms or phrases in textual description of data entities, or even filter the textual values of URIs. For instance, have at the object in the DrugBank triple:

http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00829 → http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/indication → "Used in the treatment of severe anxiety disorders, as a hypnotic in the short-term management of insomnia, as a sedative and premedicant, as an anticonvulsant, and in the management of alcohol withdrawal syndrome."

In order for a doctor to find all drugs related to the medical condition "severe anxiety disorders", a SPARQL query looking like this could be executed:

```
SELECT ?s WHERE {?s < http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/indication>
?o. Filter regex(?o, “severe anxiety disorders”)}
```

This would return the URI of the drug “Diazepam”, namely http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00829.

However, a query like the SPARQL regex query illustrated above can lead to major challenges when it comes to the query execution time, especially for large data sets. To elaborate, imagine a scenario where one is interested in finding data entities connected to the medical drugs Digoxin or Theophylline in the DrugBank data set. Digoxin is used to treat heart failure and abnormal heart rhythms (arrhythmias). It helps the heart work better and it helps control your heart rate (U.S. National Library of Medicine, 2012). Theophylline is used to prevent and treat wheezing, shortness of breath, and difficulty breathing caused by asthma, chronic bronchitis, emphysema, and other lung diseases (U.S. National Library of Medicine, 2012). If the user did not know the structure of the data set containing this information, a SPARQL query constructed like this could be executed:

```
SELECT DISTINCT ?x ?y WHERE {

{?subject ?y ?x . Filter(?y != owl:sameAs). ?x rdf:type ?type.  ?x ?property ?object. Filter
regex(?object , "\\b\\sdigoxin|theophylline\\b\\s" , "i"). Filter regex(?type , "\\bdrug\\b" , "i").}

UNION

{?x ?y ?subject . Filter(?y != owl:sameAs). ?x rdf:type ?type.  ?x ?property ?object. Filter
regex(?object , "\\b\\sdigoxin|theophylline\\b\\s" , "i"). Filter regex(?type , "\\bdrug\\b" , "i").}
}
```

This query will go through every data entity of the type “drug”, and checks whether or not the words “digoxin” or “theophylline” exist in the data resources connected to the data entity. This query would take a considerable amount of time executing, and it is highly likely that most SPARQL endpoints would return time-out the query, as it consumes a great deal of resources executing. This is mainly due to the fact that the query does not provide any specific data resources to browse through, meaning that approximately all the data entities in the data set need to be checked for regular expressions.

There are no problems with executing SPARQL filter queries as long as the user specifies in what subject URI to look for the data. However, this is often not the case, as in most queries users want to find the data entities by providing certain query input, not specifying the data entities themselves. If the users knew what specific data entities they wanted to find, there would not be much need for SPARQL filter queries. SPARQL filter queries are helpful for finding data that fulfill a certain state, and as SPARQL queries without filter clauses can only specify relationships between data entities constructed as triples, filter clauses can specify conditions that should be, or not be, met by the data entities. However, the fact that SPARQL filter queries have the tendency to execute much slower than SPARQL queries without filter clauses, the SPARQL query language is to a certain degree insufficient when it comes to retrieving information efficiently and fast.

2.2.2.2 Finding points of interest based on geo position coordinates

An important aspect of SPARQL filter queries is the filtering of numerical values. Whether one wants to find persons based on their age, geo-locations by their coordinates or weather data based on temperature, numerical filtering cannot be overlooked. Moreover, numerical filtering in SPARQL queries is a highly relevant aspect when retrieving information on the Semantic Web.

An example application is DigiTur 2, a use-case demonstrator developed at the University in Bergen for the “IKT-Norge” sponsored Sesam4 project. The application retrieved information from various data sources, including DBpedia, concerning points of interest in a specified geographic region. Since all data was retrieved via SPARQL, the results were obtained from the union of a series of FILTER queries over the relevant data sets. This introduced a processing bottleneck in the retrieval of the triples which fell in the specified region. A use-case for this thesis is therefore to improve the retrieval speed of the triples (initially from DBpedia only) which had latitude and longitude in the required region. An example of a query with the purpose of finding geo-locations based on their latitude and longitude values:

SELECT ?subject WHERE {?subject geo:lat ?lat; geo:long ?long . FILTER ((xsd:double(?lat) - 37.785834 <= 0.040000) && (37.785834 - xsd:double(?lat) <= 0.040000) && (xsd:double(?long) - -122.406417 <= 0.040000) && (-122.406417 - xsd:double(?long) <= 0.040000))}

This query filters through both the latitude and longitude values in order to find geo-locations within a certain range, covering a certain area of land. This query has to filter through a great deal of data entities and check their latitude and longitude for corresponding values. Assumedly, this query would take a while to execute in any SPARQL endpoint. This project aims at addressing how numerical filtering in SPARQL queries can be optimized.

2.2.3 Relevant literature and research

Interesting research has been conducted within the area of semantic searching and indexing of RDF data. Sindice is a lookup-index over data entities crawled on the Semantic Web (Oren et al., 2008). It is a decentralized heterogeneous data search engine for semi-structured data, such as RDF, HTML documents with RDFa tags, and Microformats or Microdata. Sindice also offers an API for developers to use in their own applications and systems. SIREn is a semantic information retrieval engine plugin to Lucene (Delbru et al., 2010), and is also the search engine which Sindice is based on.

SIREn includes a node-based indexing scheme for semi-structured data, based on the Entity-Attribute value model (Delbru et al., 2012). SIREn indexes data as tuples, storing the attributes of the entities as a sequence of elements, and offers the execution of semi-structural queries, meaning that imprecise queries containing only the local name of URIs can be executed. Further, it offers full-text queries, semi-structural queries and structural queries Moreover; SIREn makes it possible to search for data with little, partially, or much knowledge about the data itself. This is made possible by analyzing and tokenizing the data being indexed, meaning that one can execute queries consisting of partial URIs. As the Sindice project focuses mostly on storing and querying decentralized, heterogeneous data sources as a semantic search-engine on the Web of Data, FILT heads in the direction of storing and querying pre-defined data sets where the data schema is fully known. FILT does not analyze or tokenize the data being indexed so that all data values are stored as their full value, meaning that they also have to be queried by denoting their entire data values. As FILT is mainly a SPARQL filter query processing engine, this indexing approach supports the idea behind SPARQL queries, where the data schema is fully known to the user executing the query. Also, as FILT is a centralized homogeneous data store rather than a decentralized heterogeneous search-engine, as Sindice is, FILT only makes it possible to index RDF dumps, leaving out any compatibility with other semi-structural data with semantic content. Moreover, FILT does not offer semi-structural searches on the data itself, but rather focuses on replicating conventional SPARQL query executions. The approach taken by Sindice in terms of semi-structural searching is dependent on the data consisting of URIs that make sense to humans. It is easy to search for the name of a person if the predicate URIs describing this data relationship actually contains the word “name” in its local name, but if the predicate URIs are denoted as ID structures, or ambiguous or badly defined local names, semi-structural searching loses its value. FILT avoids these challenges by merely offering execution of fully structured conventional SPARQL queries that are explicitly defined by full URIs. Sindice also has different aims than FILT; as FILT mainly focuses on decreasing the query-execution time of SPARQL filter queries, Sindice additionally focuses on developing a flexible and high-performance indexing module. Sindice is dependent on its indexing mechanism to a much higher degree than FILT, as Sindice continuously indexes new triples, whereas FILT only executes the indexing process once (except from updates to the specified data set). Finally, SIREn does not offer any compatibility with SPARQL filter queries, which is the main focus of FILT.

SEMPLORE (Wang et al., 2009) also offers full-text searches through indexed RDF data. SEMPLORE treats any data value that has a data type property as a virtual keyword of concepts, meaning it will be available for full-text searches. These virtual keywords of concepts can be combined with concepts in an ontology using Boolean operators. Opposed to SPARQL queries, where a query can have multiple query targets, the querying capabilities of SEMPLORE restrict the queries to have a single query target. This supports conventional ways of retrieving information on the Web, but FILT differs from this solution in terms of letting the users query multiple targets through

SPARQL queries. Also, FILT is a database solution opposed to SEMPLORE, which is mainly a web solution.

Castillo et al. (2010) present a solution called RDFMatView for decreasing the query processing time of SPARQL queries containing multiple graph patterns. As several implemented SPARQL processors are built on top of relational databases, SPARQL queries are translated into one or more SQL queries. If queries have more than one graph pattern, the query-processing requires roughly as many joins as the query has graph patterns. Castillo et al. (2010) argue that optimizing these joins is vital in order to achieve scalable SPARQL systems. In order to avoid the computation of several join queries RDFMatView indexes fractions of queries that occur frequently in executed queries. Only graph patterns that are used together regularly in queries are indexed. RDFMatView matches FILT in terms of indexing data in order to decrease the query-execution time of SPARQL queries, but it only focuses on decreasing the query execution time of SPARQL queries with multiple graph patterns, disregarding the complications of SPARQL filter queries regarding query-execution time. RDFMatView also differs from FILT in terms of only indexing specific data based on usage patterns. As RDFMatView only indexes data based on graph patterns that are frequently executed together in SPARQL queries, FILT indexes the entire data set, treating any query equally, not depending on query statistics or usage patterns. Fletcher et al. (2008) present a similar solution to RDFViewMat, where an indexing solution for RDF data called “Three-way Triple Tree (TripleT)” has been developed, where the atoms occurring in the data set are stored independently of their roles in the data set (such as subjects, predicates or objects). The aim of TripleT is to index RDF graphs to support efficient evaluation of basic graph patterns over these graphs through SPARQL. TripleT differs from the indexing structure of FILT in terms of indexing the triple components in a random order, as FILT treats every triple of an RDF graph as actual triples in the index, meaning that the triples are stored as triples in the index. This is described in detail in section 3.1.1. Also, the aims of TripleT differs from FILT in the same way as RDFMatView in terms of the TripleT solution not being built for handling SPARQL filter queries, but rather evaluating basic graph patterns of SPARQL queries.

One of the main objectives of FILT is to reduce query-execution time of SPARQL queries containing regular expression filter clauses. Research has been conducted within the area of optimizing the regular expression filtering of SPARQL regex filter queries. Alkhateeb et al. (2009), Kochut & Janik (2007) and Lee et al. (2011) all present different methods of executing regular expressions over RDF data, but all of them uses a customized approach for executing regular expressions, whereas FILT simply executes regular expressions through the Java regular expression processing engine. Moreover, FILT does not implement new ways of executing regular expression queries, but rather executes regular expression queries through Lucene by applying the Java regular expression processing engine in the Lucene queries themselves.

There exist several solutions trying to implement efficient full-text searches through the SPARQL query language. Apache Jena LARQ (2012) is a querying solution based on Lucene and the Jena SPARQL query engine Apache Jena ARQ (2012). Nepomuk (2008) also offers the translation of full-text searches from the regex filter clause in SPARQL queries into Lucene queries. FILT differs from LARQ and Nepomuk in terms of not just implementing full-text searches, but also implementing the filtering of logical expressions and several other SPARQL filter clauses. Also, LARQ and Nepomuk do not translate SPARQL queries into customized query solutions for the users, but rather offer the possibility for the users to rewrite the queries themselves. Moreover, LARQ and Nepomuk offer extensions for performing full-text searches on literals, whereas FILT propose a solution for executing full-text searches and logical expression filtering on any triple-component through an index, directly translated from user-generated SPARQL queries.

Minack et al. (2008) present the Sesame LuceneSail solution, a part of the NEPOMUK project. Sesame LuceneSail is a solution for performing full-text search on RDF data by storing the data in a Lucene index and executing keyword queries through the index. Sesame LuceneSail is similar to the idea behind FILT, but differs greatly in certain aspects. Sesame LuceneSail executes the keyword query through the pre-stored Lucene index, and intersects the results from the Lucene query with the results of the general graph pattern SPARQL query executed in the external triplestore. This can be a costly operation if the general graph pattern SPARQL query returns large result sets. Moreover, Sesame LuceneSail includes a combined query processing where two data stores are involved in the querying process. FILT differs from this in terms of not being dependent on an external triplestore when executing SPARQL filter queries, as the general graph pattern SPARQL query stripped from filter clauses is executed over the relevant triples extracted from the Lucene query. Also, Sesame LuceneSail has certain restrictions on its query expressiveness in terms of not offering the possibility of querying more than one keyword query on each subject of a triple. FILT offers the same flexibilities and expressiveness as defined in the SPARQL query language, as FILT directly translates SPARQL filter queries into Lucene queries, obtaining the exact same results as executing the SPARQL queries through a conventional triplestore.

Many triplestores contain built-in mechanisms for coping with queries containing filtering functions. For instance, the Jena and Joseki (http://www.joseki.org/) SPARQL engines provide a possibility of executing full-text queries through LARQ. The difference between the full-text search-engine in LARQ compared to FILT is that LARQ requires the SPARQL queries to include different syntaxes that do not correspond with the general SPARQL syntax. FILT does not require any additional statements or functions in the SPARQL queries and executes regular SPARQL queries with filter clauses. Full-text searches through FILT are simply run by adding a regex filter clause in the SPARQL query based on the standard SPARQL syntax. Another example of a built-in mechanism for executing specific filtering functions is the SQL MM function for executing geospatial queries in the Virtuoso triplestore (http://virtuoso.openlinksw.com/). The SQL MM function in Virtuoso makes it more efficient to execute geospatial queries (OpenLink Software, 2009). However, just as Joseki and Jena combined with LARQ, the built-in SQL MM filtering function in Virtuoso is dependent on another query-syntax than SPARQL filter queries, meaning that the SPARQL queries have to be modified from their original syntax in order to benefit from the built-in filtering mechanisms. FILT is not dependent on additional filter statements or different query syntaxes in order to execute filter queries, as FILT is not database-specific and are compatible with any conventional triplestore.

2.2.4 Research questions and success criteria

As the main objective in this project is to decrease the time spent on executing SPARQL filter queries, it is possible to solely focus on implementing tools that would speed up the query execution time. Based on this, it is natural to look into the field of indexing data, as this is a quick and lenient way of retrieving data based on matching expression, terms and phrases. There exist several frameworks for indexing data, and many of them are open-source and free. An example of such a framework is Apache Lucene (Apache Lucene, 2011), which has been frequently implemented in many research projects throughout the years, as the previous section presented. Based on the fact that Lucene is a commonly implemented tool in research projects and industry-standard SPARQL processing engines, this project will also implement the Apache Lucene framework.

The research questions that this project intends to answer are:

- Can the query-execution time of SPARQL filter queries be decreased by storing the RDF data and executing the SPARQL filter queries through the Apache Lucene framework? If so, what are the time differences of the SPARQL filter queries compared to conventional RDF stores?
- How can RDF data be stored through the Apache Lucene framework in order to most efficiently retrieve RDF data from SPARQL filter queries?
- In what way must the filter expressions of SPARQL filter queries be re-written in order to utilize the possibilities, and cope with the restrictions, of the querying module of the Apache Lucene framework?
- How can the built-in query library of the Apache Lucene framework support the execution of the regex and logical/numerical expression SPARQL filter clauses?

The success criteria that should be fulfilled by the system in order to meet the research questions are:

- All general SPARQL queries without filter clauses, as well as SPARQL regex- and logical/numerical expression filter queries containing simple graph patterns and filtering, should be executable through the system

- SPARQL regex- and logical/numerical expression filter queries should execute faster through the system than through a conventional triplestore
- All the results returned from SPARQL queries should be returned in the same format as a conventional triplestore

In this project, a new way of storing and querying RDF data called FILT (Filtering Indexed Lucene Triples) has been built based on the Apache Lucene framework. This text will shed light on the purpose and achievements of the solution, as well as the challenges and issues faced by developing such a system. However, before presenting the FILT solution, the text will first present an analysis of relevant literature within the field of optimizing SPARQL filter queries.

Chapter 3: Implementation

The idea behind FILT was to create a mechanism for storing and retrieving data expressed as RDF triples. It was determined that the solution should be built on top of the Apache Lucene framework. This decision had its foundation in the idea of implementing an efficient high-level, platform independent information retrieval tool which would demand little CPU and RAM usage. Due to the fact that there were limited funds and time allocated to the research project, it was important that FILT should be built on top of an already existing indexing tool that was free, platform-independent and open source. There already exist indexing frameworks that meet these criteria, meaning that there was no need for creating a whole new storage and retrieval mechanism for indexed data. Instead, FILT could solely focus on implementing a specific solution for storing and retrieving RDF data built on top of an already existing indexing framework.

The Apache Lucene framework was found to be the most relevant framework based on the criteria of having a free and open license, and being platform-independent. It is also the most commonly used indexing framework both on the web and in internal information systems. Hence, it was decided that FILT should be built from scratch, depending solely on the Apache Lucene and Jena libraries for indexing and retrieving data.

FILT is a SPARQL filter processing engine and enables storing and querying of RDF data through the Apache Lucene framework. Its main purpose is to decrease the query-execution time of SPARQL queries containing filter clauses, thus optimizing the efficiency of semantic information retrieval. FILT currently provides storing of triples, a SPARQL endpoint, and a SPARQL querying user-interface. FILT can store any data set stated as triples. The data set must be expressed in one of the three most common syntaxes for triples: N-Triples, Turtle or RDF/XML. Moreover, FILT will supplement a traditional triplestore by stripping filter queries away from the SPARQL query during a pre-processing phase. It then passes the set of triples that match the filter conditions back to the Jena SPARQL query engine. General SPARQL queries without filter clauses will be sent directly to an external triplestore SPARQL endpoint, or to a local RDF model of the entire data set. This means that a SPARQL endpoint URL of a triplestore, or the raw RDF data set file, has to be specified in FILT in order for any type of SPARQL query to run properly.

The architecture of FILT is shown in Figure 3.1. This figure illustrates how SPARQL queries are executed through FILT. There are several steps in this process: first the user issues a SPARQL query. If the query does not contain filter clauses, the query is immediately executed through an external RDF store, either a triplestore or a local RDF model loaded into the Jena framework. If the SPARQL query contains filter clauses, it is sent to the query rewriting module which performs two processes:

extracting the filter clauses from the query and transforming them into Lucene queries, and stripping the filter clauses from the SPARQL query, leaving only the general SPARQL query. The general query will be stored for a later use with the conventional triplestore, as FILT is only built to cope with the filter clauses of SPARQL queries. The Lucene queries constructed based on the filter clauses in the query are executed through the Lucene index consisting of the indexed data of the entire RDF data set. This index must be generated prior to the querying process by specifying to the system one or more data set files that should be indexed. Further, the output of the Lucene queries executed through the index consists of triples that will be the foundation of building an internal RDF model. This data model now contains the triples corresponding to the filter clauses of the SPARQL query, and the general SPARQL query stripped of filter clauses will be executed over this local model. Finally, the output returned from the general SPARQL query is the final query output that will be returned to the user that issued the SPARQL query.

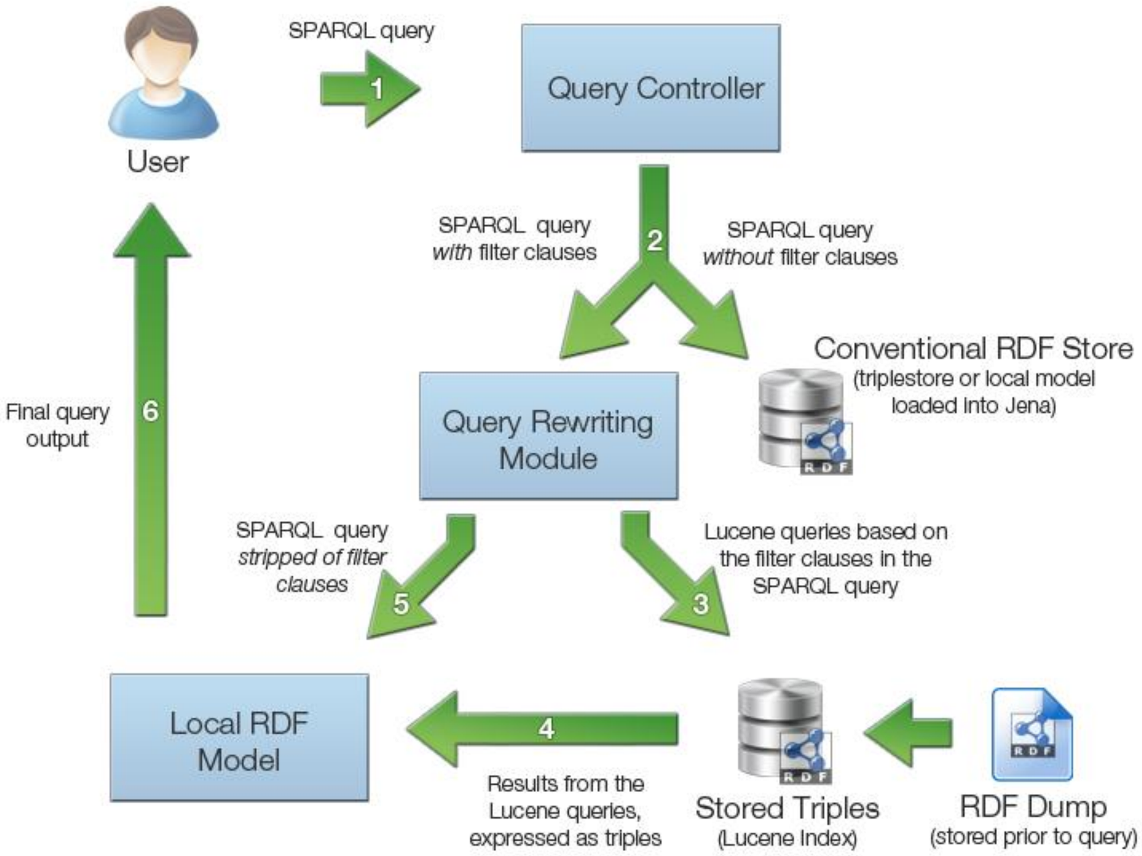


Figure 3.1: The architecture of FILT

This section will further describe all the aspects of FILT in detail. Initially, the index structure and indexing process will be described. Next, the query rewriting module will be highlighted, before elaborating on the actual querying process. Finally, further implementations and issues that occurred during the development of FILT will be discussed.

3.1 Indexing RDF data with the FILT framework

The indexing process in FILT is a specific indexing solution based on the Lucene libraries for indexing textual and numerical values. FILT is dependent on Lucene indices in order to retrieve data from a given data set, meaning that the RDF data has to be indexed in order to query it. FILT offers an indexing module which will convert the raw RDF data into a compatible index structure, meaning that the index converts all the raw triples in a given RDF graph into a Lucene index format. This section will portray the entire indexing module of FILT,

3.1.1 Index structure

This section will elaborate on the process of structuring Lucene indices constructed from raw RDF, which was a vital activity in this project. As mentioned in section 2.1.3.1, Lucene indices consist of documents containing fields. A Lucene document in this context can be viewed as one specific entry in the index, where the fields of the document contain all the data or information that is related to each other. Moreover, a document can be viewed as a data entity. This is a useful perspective when dealing with RDF data, as RDF data consist of graphs, where data resources are bound by given relationships with each other. In this perspective, Lucene documents and RDF graphs are very similar, as they both consist of one or more entry that represents relationships to other data resources.

In order to understand the fundamental index structure of Lucene, it can be helpful to once again have a quick look at how queries can be executed over such indices. As mentioned in section 2.1.3.2, the basic querying structure in Lucene is built on specifying a document field to look up and a value to match the data in the given document field. For instance, one could have indexed a document containing a field with a field-name “title” and a field-value “Apache Lucene”. In order to find this specific document, a query looking like this could be executed: title:Apache Lucene. This would find all the documents corresponding to a title field with a value “Apache Lucene”.

During the process of developing an indexing mechanism optimized for querying RDF data, several different index structures were implemented before eventually coming up with the most efficient structure. This was partially due to the fact that it was desirable to test different solutions for indexing RDF data, as well as the fact that the development was a learning-process and that most questions were answered during the course of the project.

3.1.1.1 Implementation #1

The first index structure to be implemented was based on a commonly-used structure for indexing web documents. It is frequently used in web-based information retrieval systems such as search engines, and simply consists of two document fields: title and content. In the context of indexing RDF data, this specific indexing structure would index the subject of the given RDF graph in one field, and all the predicates and objects associated with that subject in another field, meaning that instead of naming the two document-fields “title” and “content” they would rather be named “subject” and “graph” (see Table 3.1). Moreover, this means that there are two field-names and two field-values that can be looked up, namely the subject-URI and its RDF graph. In the light of how SPARQL filter clauses are constructed, this way of structuring the index would lead to inefficient ways of executing Lucene queries transformed from SPARQL filter clauses. This is due to the way SPARQL filter clauses are constructed, as they are commonly composed of a triple-object variable that should correspond to a given value. With the given index structure, it would not be possible to look up this variable in an efficient way as there does not exist separate fields for each triple, meaning that one would have to look up the entire RDF graph in order to find the given predicate and object. This does not only lead to complications in terms of retrieving the correct data output from the queries, but also querying the correct data. Since the entire RDF graph was stored in one field, it would in most cases be impossible to query the value of one predicate, and not another. This means that in order to query only one specific predicate or object in the RDF graph, one would first have to retrieve the entire RDF graph as a text string and then execute the SPARQL filter clause value through regex matching on the RDF graph. This would undermine the fundamental idea of this project and the efficiency of querying Lucene-based indices. Furthermore, this would be a time-consuming process, thus contradicting the entire purpose of building FILT, namely the optimization of SPARQL query run-time. This implementation was far from fulfilling the objectives of FILT, because the indexing architecture led to slow and inefficient querying, and it was difficult to derive specific information from the fields when returning results from the queries. This meant that another index structure had to be implemented.

Table 3.1: The initial index structure of FILT

Field-name	Field-value
subject	The subject-URI of a given RDF graph
graph	The RDF graph of a given subject-URI

3.1.1.2 Implementation #2

Based on the outcome of the first proposed suggestion for how the indices in FILT should be structured, it was obvious that there was a need for a more accessible structure that could be efficiently queried through Lucene, without having to perform lookup operations on the query output after executing a given query. The most evident way to do achieve such a thing would be to implement more fields in every document that was indexed, providing additional possibilities of accessing the data through queries, and also retrieve the relevant output based on the queries. When adding more fields to the index structure it would be sensible to consider how RDF data is structured. Since RDF data is made up by triples, the most evident way of storing the data would be dividing the triples into three separate document-fields, namely subject, predicate and object (see Table 3.2). Moreover, this means that for each RDF graph, representing a subject and its related triples, there would be indexed one document containing separate fields for storing the subject-URI and all the predicates and objects from the triples where the subject-URI acts as the subject. This can be illustrated in a more formal way:

```
for each sub-graph in the superior graph {
    new Document
    add field to document(FieldName: subject, FieldValue: <subject-URI>)
    for each predicate and object in graph {
        add field to document(FieldName: predicate, FieldValue: <predicate URI>)
        add field to document(FieldName: object, FieldValue: <object value>)
     }
}
```

Table 3.2: The index structure in implementation #2

Field-name	Field-value
subject	The subject-URI of a given RDF graph
predicate	The predicate URI of a given triple
object	The object-value of a given triple

Compared to the index structure described in the previous section, this solution would offer more efficient ways of querying and retrieving data. This is due to the fact that the data is now separated into more fields, thus making the data more accessible. Instead of being left with the restriction of having to retrieve the entire RDF graph in every query, one could specify predicate URIs and object values in the Lucene queries. To illustrate the querying process, consider this SPARQL query:

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?subject WHERE { ?subject rdfs:label ?label. Filter(str(?label) = 'Car') }

This query aims at retrieving a subject-URI (?subject) that is a part of the triple consisting of the subject-URI, the predicate URI "http://www.w3.org/2000/01/rdf-schema#label" and an object-value that should correspond to a String filter with the value "Car". Based on the current indexing structure, this could be queried through Lucene by composing a query looking like this:

predicate: http://www.w3.org/2000/01/rdf-schema#label object:Car

This query would retrieve all the relevant documents containing the document field "predicate" with the value "http://www.w3.org/2000/01/rdf-schema#" and the document field "object" with the value "Car". However, there is one major issue with this query, which has its foundation in the way the index is structured: the query will only check if a document contains the given predicate and the given object-value. What it does not take into consideration is if these two values are related to each other in any way. It merely checks if the predicate exists in the RDF graph, and if the object-value exists in the RDF graph. This means that the index structure makes it impossible to know if the object-value is related to the predicate specified in the query, or if it is related to any of the other predicates in the document. Moreover, the index structure only keeps hold of the two values isolated from one another, rather than keeping track of a possible connection between the two. Considering the previous query example, this index structure will lead to imprecise query results, as the some index documents may contain the predicate URI "http://www.w3.org/2000/01/rdf-schema#" and the object-value "Car", but the object-value could be the value of another object, rather than the object related to the given predicate in the original RDF graph. Consequently, this is not an optimal way of indexing RDF data, as the index structure disputes against the principles in querying RDF data. This means that another index structure had to be implemented in order to make the querying process more suited for RDF data.

3.1.1.3 Implementation #3

Despite the second index structure implementation being inadequate, it was evident that the index structure implementation was fundamentally close to being a sufficient solution, based on the fact that it stored all the components of every triple in the RDF graph that was indexed. However, there was a need for an index structure that would make it possible to query every object-value of a triple efficiently and accurate, opposed to the second implementation where it was impossible to know what triple the object-value one queried was a part of. In order to achieve this there would have to be implemented an efficient way of looking up every predicate and its object-value. In order to look up a specific value in Lucene, one would have to know what document-field that value is stored in, meaning that one can look up specific data or information by specifying document-field names. When looking at the second implementation of the index structure, the architecture merely let one look up if a given predicate and object-value existed isolated from one another. This was because all the predicates and objects of the RDF graph were stored in the document-fields named "predicate" and "object", meaning that they were all stored in fields with the same lookup-name. This made it impossible to perform unique look-ups, and one could only know if a certain predicate and object existed in the index, ignoring what relations they had to other data resources.

Based on the fact that Lucene is based on a lookup mechanism where the names of the document-fields act as an identifier for finding specific data or information, the third implementation is based on an index structure with dynamic document-fields. Instead of having document-fields named "subject", "predicate" and "object", the index structure would combine the two latter document-fields into one field, by naming the field the predicate URI and giving it the object-value of the given triple as its input. Moreover, this means that apart from the static field named "subject", the other document-field names will vary depending on what the predicate URI is (see Table 3.3). Also, all the documents contain a field with the field-name “graph” and a field-value containing the filename of the data set being indexed. If several data sets contain data about entities already indexed, the graph field will have multiple values referring to the each of the data sets. This is implemented in order to making it possible to retrieve what RDF graph the results of a query are retrieved from. The overall index structure can be described in a more formal way like this:

```
for each sub-graph in the superior graph {
    new Document
    add field to document(FieldName: graph, FieldValue: <The filename of the data set file>)
    add field to document(FieldName: subject, FieldValue: <subject-URI>)
    for each predicate and object in graph {
        add field to document(FieldName: predicate, FieldValue: <object-value>)
    }
}
```

To illustrate this further, imagine a sub-graph of a superior RDF graph looks like this:

```
ns:Pneumonia
        rdf:type  owl:Thing , ns:Symptom ;
        rdfs:label "Pneumonia"@en , “Lungebetennelse”@no ;
        ns:isSymptomOf dbpedia:Allergic_bronchopulmonary_aspergillosis;
        ns:isSymptomOfBodyPart ns:Lung ;
       owl:sameAs dbpedia:Pneumonia .
```

In the current index structure, the document describing this data entity would look like the structure of Table 3.3. There is an important note to take from Table 3.3 regarding the storage of the triple ns:Pneumonia rdfs:label "Pneumonia"@en, “Lungebetennelse”@no. The index structure will take any object-value with a language tag and store them in separate fields. This makes the data more accessible through queries, and also easier to retrieve the correct value of the triple as output. However, this will increase the disk space consumed by the index, as there will be stored several more document-fields compared to merely storing all the literals of different languages in the same field. On the other hand, this would reduce the efficiency of querying to a great extent. Hence, the decision was made to rather have a slightly larger index that was more efficient for querying, compared to having a smaller index that would not be as efficient for querying the data.

Table 3.3: The final index structure implementation of FILT

Document-field name	Document-field value
graph	<The filenames of the data set files>
subject	ns:Pneumonia
rdf:type	owl:Thing, ns:Symptom
rdfs:label-en	Pneumonia
rdfs:label-no	Lungebetennelse
ns:isSymptomOf	dbpedia:Allergic_bronchopulmonary_aspergillosis
ns:isSymptomOfBodyPart	ns:Lung
owl:sameAs	dbpedia:Pneumonia

The third index structure implementation is the final and current index structure version of FILT. All the document-fields in the index have the attributes “Index.NOT_ANALYZED_NO_NORMS” and “Store.YES”. These attributes are described in section 2.1.3.1. The fields are not analyzed because FILT is not based on finding closely-related data entities based on a search term. FILT matches input from users through SPARQL filter clauses against indexed RDF graphs, meaning that all the data in the indexed RDF graphs must have the same structure as the raw RDF data in order perform correct queries. If the document-fields were analyzed, the values of these fields would be split into different terms that would make it easy to find data entities based on vague search terms, which is not the purpose of FILT. All the document-field values are stored in the index, meaning that they are retrievable as output from queries. This increases the disk-space consumed by the index, but is vital for providing output from the SPARQL queries being executed.

The document fields of the index are stored as two different field types, based on the data type of the value that should be stored in the document field. Numbers are stored in a “NumericField” in order to execute numeric Lucene queries that can match number ranges. Ordinary document fields cannot match numeric ranges. All data values that are not numbers are stored as the ordinary document field “Field”.

3.1.2 Indexing process

The indexing process of FILT consists of several mechanisms for converting, mapping and storing data. The process is constructed to convert raw RDF data into a Lucene index format. This section will describe all the steps and aspects of the indexing process of FILT in detail.

3.1.2.1 Stage 1: Pre-processing

FILT makes it possible to index any RDF graph written in the RDF/XML, Turtle and N-Triples format. Before running the indexing process, one should specify the absolute path to the RDF data set file to be indexed, as well as the absolute path to the folder one wants the index to be stored in. When this is defined the indexing process can be executed. The first aspect of the indexing process is for the system to determine what language syntax the RDF data set file is written in. This is done because the architecture of the indexing process is built on indexing data of the N-Triples format, meaning that if the data are stated in RDF/XML or Turtle, the data will be converted to the N-Triples format before further processing the data. The format of the data set file is determined by analyzing the structure of first statement in the file, as the syntax of the statements will differ between the different syntaxes. If the data is written in RDF/XML or Turtle, the data is converted to the N-Triples format and written to a new file which is stored in the same location as the original data set file. The temporary data set file generated in the N-Triples format will be the file that the indexing process read in order to index the RDF data. This temporary file will be deleted when the indexing process is finished. Moreover, the requirements regarding the initial stage of the indexing process are:

- Specifying an RDF data set file written in the RDF/XML, Turtle or N-Triples format
- If the RDF data set file is written in RDF/XML or Turtle, there should be available a minimum disk space twice the size of the original data set file during the entire indexing process, as there will be a temporary data set file of the N-Triples format written to the disk (the N-Triples format consumes more disk-space than the other two formats)

When the process of determining the format of the data set file and making sure there is a data set file of the N-Triples format available, the process then establishes if an index already exists in the absolute path to the specified folder where the index should be stored. If an index already exists, the data stored in that index is loaded into the system before any other data will be processed. This provides the possibility of indexing several RDF graphs into the same index, meaning that one can add more data to the index at a later stage.

3.1.2.2 Stage 2: Reading and processing the RDF data

The actual process of reading the RDF data is executed by reading the data set file line by line. Based on the index structure described in section 3.1.1.3, the system relies on mapping the entire RDF graph of every subject-URI in the data set prior to the actual storing of index documents. This means that before any index documents are being stored, the indexing process reads through the entire data set file and maps together all the subject-URIs and their RDF graph in a local data structure. This is necessary due to the fact that RDF data somewhat has a "chaotic" structure, meaning that triples can be stated randomly wherever in the document. For instance, a subject in a triple stated in the first line of the data set file can also be the subject in a triple stated in the last line of the same data set file. This means that in order to fulfill the index structure of indexing every sub-graph in the larger RDF graph as one specific document, the system has to read through the entire file and make sure that all the subjects of the triples and their RDF graphs are mapped together.

The Lucene framework provides a way of dealing with this mapping real-time instead of mapping data prior to the actual indexing process by offering an indexing method called "updateDoc". This method will make it possible to look up a document and edit the information stored in that document. This method was tested during the implementation of FILT, and it had several complications, such as the indexing process consuming much more time, as well as the index itself taking up significantly more disk space. Also, it was hard to modify the already stored fields in the index, and to fulfill the architecture of dynamic fields and grouping RDF graphs together. Due to this, the idea of using the "updateDoc" method was discarded.

The mapping of the RDF graph is stored in a HashMap with a key of the data type String (the subject of the RDF graph), and a value of the data type HashMap, which again has a key and value of the data type String (the predicates and objects of the RDF graph). More formally, the data structure of the HashMap looks like this:

HashMap<String, HashMap<String, String>

To illustrate the mapping process further, have a look at the sub-graph of a larger RDF graph describing the data entity ns:Pneumonia, presented in section 3.1.1.3. This graph would be stored in the HashMap like this:

{ns:Pneumonia={p1:type=p2:Thing, p3:Symptom, p4:label-en=Pneumonia, p4:label-no=Lungebetennelse, p3:isSymptomOf=p5:Allergic_bronchopulmonary_aspergillosis, p3:isSymptomOfBodyPart=p3:Lung, p2:sameAs=p5:Pneumonia}}

As shown in this example, the mapping of the RDF graph also includes replacing the URIs specified in the graph with local prefixes. These local prefixes are generated for all the URIs in the data set file with the single purpose of reducing the disk space consumed by the index itself. To illustrate this, imagine a data set with the size of 250 megabytes being indexed. This data set is likely to consist of tens of thousands data entities, meaning that there will be even more triples, most likely in the region of hundreds of thousands. When storing all of these triples in an index, the index would consume much more disk space if every full URI were stored, compared to replacing all of these URIs with a much smaller local prefix. For instance, instead of storing the full URI of the predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" the indexing process can replace the namespace URI with a local prefix like this "p1:type". The local prefix is generated by the letter "p", representing the word "prefix", along with an integer. This integer is being incremented every time a new unique namespace URI is located in the data set. This means that if there are 100 unique namespace URIs, the last generated local prefix will be "p100". The local prefixes are reducing the characters needed to be stored for each URI greatly, and when applying this for hundreds of thousands triple-components the disk space consumed by the index will be reduced to a great extent. During the process of converting URIs to a local prefix, all the full URIs are mapped to its local prefix and written to a file at the end of the indexing process. This file is loaded back into the system whenever the FILT querying engine is initialized. This way, all the URIs defined in the SPARQL queries can easily be translated into the local prefixes by looking them up in the map containing the full URIs and their local prefixes. This lookup has to be reflexive, meaning it is necessary to have the possibility of looking up both the full URI when the users specify such URIs in the SPARQL queries, and looking up the local prefixes when retrieving output of the queries from the index. The output from the index will naturally contain the local prefixes instead of the full URIs, which has to be converted to the full URIs before displaying the output to the users. The local prefixes are only applied to reduce disk space of the index itself, and will not make any sense to provide as output to the users. For output in the form of literals such a conversion will not take place, as literals are not built up by URIs.

As mentioned in the previous section, it is possible to index several RDF data set files isolated from one another. In order to this, the mapping of all the data resources has to be loaded back into the system in order to continue the mapping process. This is done in order to make it possible to index triples containing a subject that has already been indexed. In order to map the new triples to this subject, the already existing mapping structure has to be loaded into the system.

3.1.2.3 Stage 3: Indexing Lucene documents based on the RDF data mappings

The third and final stage of the indexing process includes the storing of the mapped RDF data as Lucene documents. The index structure is based on the index implementation presented in section 3.1.1.3. This index structure is generated based on the pre-processed RDF data stored in the data resource map. The algorithm for doing this looks like this:

```
Iterator<String> it = resourceMap.keySet().iterator();

       while (it.hasNext()) {
              String subject = it.next();
              HashMap<String, String> predicateObjectMap = resourceMap.get(subject);
              addIndexDocument(input.getName(), subject, predicateObjectMap);
              }

       writer.close();


public void addIndexDocument(String graph, String subject, HashMap<String, String>
predicateObjectMap) throws CorruptIndexException, IOException {

              Document doc = new Document();
              doc.add(new Field(IndexDataSpecific.graphField, graph, Store.YES,
              Index.NOT_ANALYZED_NO_NORMS));
              doc.add(new Field(IndexDataSpecific.subjectField, subject, Store.YES,
              Index.NOT_ANALYZED_NO_NORMS));

              Iterator<String> it = predicateObjectMap.keySet().iterator();
              while (it.hasNext()) {
                     String predicate = it.next();
                     if (!languageList.contains(predicate)) {
                            if (NumberUtils.isFloat(object)) {
                                   float objectFloat = Float.parseFloat(object);
                                   NumericField nf = new NumericField(predicate,
                                   Store.YES, true).setFloatValue(objectFloat);
                                   doc.add(nf);
                                   }
                            else {
                                   doc.add(new Field(predicate, object, Store.YES,
                                   Index.NOT_ANALYZED_NO_NORMS));
                            }
                     }
              }
              writer.addDocument(doc);
              writer.commit();

       }
```

The overall size of the index folder after the indexing process has finished will be approximately the same size as the raw data set. The index itself is not as big as this, but the data resource map containing the all mappings of the RDF data constitutes to about half the size of the index folder. This resource map is necessary to store in order to making it possible to index other RDF graphs to the same index.

3.1.3 Restrictions with the indexing RDF data through FILT

There are various limitations with the current architecture of FILT when it comes down to the indexing process itself. First of all, due to the fact that all the pre-processed data mappings are stored in the memory of the system, it means that the process will demand a great deal of temporary memory in order to run. It is possible to allocate memory to the application in order for it to not go out of memory, but for huge data sets the system will most likely run out of memory at some point.

Another important aspect to mention regarding the indexing process is the fact that it is not possible to stop the indexing process and continue it at a later stage. Moreover, this means that when starting the indexing of a given data set, one must wait for the indexing process to finish in order for the data to be indexed. If the index process is terminated before it is finished, the index process has to be executed all over again. This is due to the fact that in order for the indexing process to be continued, the resource map containing all the mapped RDF data would have to be written to a file after each time a new sub-graph was added to the map. This would be necessary in order for the process to load all the already mapped data back into the system. However, this would be time consuming and error prone, as the user might stop the process while the data resource map is written to a file, meaning that the data could be faulty and not readable the next time the process was continued. Because of these complications, it was found most purposeful to seclude this aspect from the system and rather focus on developing the system according to the research questions of the project.

It is also worth mentioning that the indexing process may have complications with indexing huge data sets written in the Turtle syntax. This is because the system will try to convert Turtle data sets to the N-Triples format, meaning that it is a possibility that large data sets will cause the system to run out of memory due to the fact that the data is loaded into the internal memory of the system in order to convert the data. Based on this, it is recommended to primarily index data sets written in the N-Triples format, unless the data sets are of a small size.

3.2 Rewriting SPARQL queries to Lucene queries

This section will present the most important aspects of the query rewriting module in FILT. This module serves several purposes, such as rewriting queries to match the desired syntax required by the solution, extracting and mapping filter clauses, analyzing and converting namespaces and prefixes of SPARQL queries to correspond with the index, and obtain necessary data to retrieve the correct output of the SPARQL queries. All of these features will be presented in this section.

3.2.1 Manipulating the SPARQL query strings

A small, but important, aspect of the query rewriting module in FILT is to make sure the SPARQL queries given as input to the system contains the syntax and structure the system is built to operate with. Before the queries are being analyzed and processed, the system executes several operations related to rewriting the query strings to a desired structure and syntax. This is done in order to achieve a standard way of performing further operations on the query. For instance, the query syntax should be based on the n-triples format, meaning that all the triples in the SPARQL query should explicitly be stated as subject → predicate → object, compared to the Turtle syntax which allows for the same subject of several triples to merely be defined once, with the combination of predicates and objects being separated by semicolons. An example of a simple SPARQL query based on the Turtle syntax can look like this:

*SELECT * WHERE {?s rdfs:label ?label; geo:lat ?lat; geo:long ?long.}*

This query will be rewritten to match the n-triples format like this:

*SELECT * WHERE {?s rdfs:label ?label. ?s geo:lat ?lat. ?s geo:long ?long.}*

This rewriting has a single purpose of making it easier to analyze and extract information from the query. The query rewriting module also makes sure the query consists of a pre-defined template of words as a substitute for query terms that are not of the desired format. For instance, this includes the re-writing of some lowercase words to uppercase words and vice versa. To illustrate this feature, imagine the SPARQL query described above, with a filter clause added to it, looking like this:

*select * where {?s rdfs:label ?label; geo:lat ?lat; geo:long ?long. FILTER regex(?label, “Norway”)}*

This query would be rewritten into this format:

*SELECT * WHERE {?s rdfs:label ?label. ?s geo:lat ?lat. ?s geo:long ?long. Filter regex(?label, “Norway”)}*

This transformation serves the single purpose of making it easier to create a standard way of analyzing the query at a later stage.

3.2.2 Mapping the filter clauses of SPARQL queries

The query rewriting module in FILT primarily includes the extraction of filter clauses from the SPARQL queries that are executed. In order to know what index field to query, what type of query that should be executed and the value of every filter clause, it is necessary to analyze and extract information from the SPARQL query. This is done by extracting every filter clause from the SPARQL query and put them in a map along with what index field (predicate) that should be filtered. This map will be referred to as the filter map. The data structure of the filter map is HashMap<String, HashMap<String, ArrayList<String>> where the predicate is the key, and the value is a HashMap containing the filter clause type as a key and its filter values stored in a list. To illustrate this, imagine a SPARQL query constructed like this:

SELECT ?label WHERE {?s rdfs:label ?label; owl:sameAs ?sameAs; rdf:type ?type. Filter regex(?label, 'de\\Z'). Filter(?sameAs != dbpedia2:Omalizumab). Filter(?sameAs != dbpedia2:Pancrelipase). Filter (?type != dbpedia2:references) }

This query is constructed to execute over the DrugBank data set (http://www4.wiwiss.fu-berlin.de/drugbank/). It aims to find the rdfs:label of every data entity that has an object connected to the rdfs:label predicate that ends with the letters “de”, where the object connected to the owl:sameAs predicate does not equal dbpedia2:Omalizumab or dbpedia2:Pancrelipase, and finally where the object connected to the “rdf:type” predicate does not equal dbpedia2:references. The first step in the process of rewriting this query will be to extract its filter clauses as a single text string. In this example, the filter string would look like this:

Filter regex(?label, 'de\\Z'). Filter(?sameAs != dbpedia2:Omalizumab). Filter(?sameAs != dbpedia2:Pancrelipase). Filter (?type != dbpedia2:references)

For each filter clause in the filter string, the system finds the type of the filter clause present, the filter value of the filter clause, and the predicate variables present in the filter clause. These values are put in the filter map previously presented in this section. The process of mapping the filter clauses can be exemplified by having a look at the filter string extracted from the SPARQL query. The loop will treat each filter clause in the filter string like this (note that these queries are being demonstrated by specifying the SPARQL query predicate in order to make it more intuitive in this text, whereas FILT actually applies internally generated prefixes in order to query the index. See section 3.1.2.2 for more details):

1. *Filter regex(?label, 'de\\Z') →{rdfs:label={regex=[de\\Z]}}*
2. *Filter(?sameAs != dbpedia2:Omalizumab) →{owl:sameAs={logicalexpression = != dbpedia2:Omalizumab}}*
3. *Filter(?sameAs != dbpedia2:Pancrelipase) →{owl:sameAs={logicalexpression =[!= dbpedia2:Omalizumab && != dbpedia2:Pancrelipase]}}*
4. *Filter (?type != dbpedia2:references) →{rdf:type={logicalexpression =[!= dbpedia2:references]}}*

In step 1, there is a mapping consisting of the key “rdfs:label”, the predicate that should match the field in the index, the filter clause “regex”, which will define what type of Lucene query to build, and the filter value “de\\Z”, which will be the value of the Lucene query. Step 2 defines a mapping represented by the predicate key “owl:sameAs”, the filter clause type “logical expression”, and the filter value “!= dbpedia2:Omalizumab”. The “owl:sameAs” predicate is left out of the filter value, as the predicate is already stored as the key of the map. Step 3 is the most interesting step in terms of understanding the entire filter mapping process, as the third filter clause contains the same predicate and the same type of filter clause as in the second filter clause. The third filter clause is therefore put into the already existing instance of the predicate key “owl:sameAs” that was defined in step 2. The filter clause type in the third filter clause is also the same as the filter clause type in step 2, meaning that the filter value has to be combined with the already existing filter value assigned to the filter clause type. Moreover, since both the predicate key and the filter clause type key are the same in the second and the third filter clause, their values must either be combined with each other, or added as separate values in the ArrayList<String> list value (the filter value list) of the HashMap<String, ArrayList<String> (the map with a filter clause key and a filter value list) in the final filter map. In this example, the filter values are being combined instead of added as separate entries in the filter value list. This is done to illustrate that FILT can analyze logical expressions, meaning that the filter values can be separated by logical operators such as AND (“&&”) and OR (“| |”). All the different filter clause types and their functions are described in more detail in section 3.3. If a filter clause value containing logical expressions separated by the AND operator, the expressions can either be added as separate entries or as one entry separated by the AND operator. They can be added as separate entries due to the fact that expressions separated by the AND operator *must* return true in order for the entire expression to return true. Adding them as separate entries will serve this purpose the same way as separating the expressions by the AND operator. However, expressions separated by the OR operator has to combined into one filter value expression. This example separates the filter values by the AND operator to illustrate the different aspects of the system. Step 4 has a same mapping structure as step 2, only with a different predicate and filter value. The four different steps of mapping the filter clauses will result in a final filter map looking like this:

{rdfs:label={regex=[de\\Z]}, owl:sameAs={logicalexpression =[!= dbpedia2:Omalizumab && != dbpedia2:Pancrelipase]}, rdf:type={logicalexpression =[!= dbpedia2:references]}}

This map will be the foundation for building different types of Lucene queries that will be executed through the index. The querying module of FILT is presented in section 3.3 and will not be described in this section.

3.2.3 Managing namespaces and prefixes

As mentioned in section 3.1.2.1, when indexing RDF data with FILT the system operates with the N-Triples format, regardless of the format of the input file specified by the user. This means that the indexing process reads the data set line by line, and that every triple is explicitly defined with a subject, predicate and object, opposed to the Turtle format where it is only necessary to specify the subject at the beginning of its RDF graph (Becket & Berners-Lee, 2011). As mentioned in section 3.1.2.2, the full URIs of every namespace referred to in the input file are mapped to local prefixes that will be stored in the index instead of the full URIs. This is mainly done to minimize disk storage use of the index itself. However, this poses a challenge when executing SPARQL query through FILT, due to the differences between the local prefixes in the index and the query prefixes in the SPARQL query. It is necessary for the system to convert the prefixes specified in the SPARQL query to the local prefixes stored in the index, and finally convert the local prefixes of the query output retrieved from the index to the full URIs of each prefix. To elaborate on this, there are three different namespace maps, each of them storing a mapping needed to fulfill the operations of querying the correct prefixes and specifying the valid query output. The first map is created in the indexing process and contains a mapping of the full URIs as keys and the local prefixes generated in the process as values. Another map is also created during this process and contains a reverted relationship between the full URIs and the local prefixes, meaning that the local prefixes act as keys in the map and the full URIs are the values of these keys. The final map is a dynamic map that is created every time a SPARQL query is executed through the system. This map is based on an analysis of the prefixes defined in the query itself, and maps the query prefixes as keys and the full URI of each prefix as the value. Moreover, we get three maps with these formal mappings:

- URI → local prefix
- Local prefix → URI
- Query prefix → URI

These maps combined offer the possibility of looking up the desired prefix or URI needed to perform different operations. To illustrate this, imagine we have indexed a data set resulting in the full URI to local prefix map and the local prefix to full URI map looking like this:

- URI → local prefix: *{http://www.w3.org/1999/02/22-rdf-syntax-ns#=p1, http://www.w3.org/2000/01/rdf-schema#=p2, http://www.w3.org/2003/01/geo/wgs84_pos#=p3, http://www.w3.org/2001/XMLSchema#=p4}*
- Local prefix → URI: *{p1=http://www.w3.org/1999/02/22-rdf-syntax-ns#, p2= http://www.w3.org/2000/01/rdf-schema#, p3=http://www.w3.org/2003/01/geo/wgs84_pos#, p4=http://www.w3.org/2001/XMLSchema#}*

If a SPARQL query was executed through the system at this stage, the first process would be to map the query prefixes. This can be demonstrated by executing this SPARQL query:

PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#=geo>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#}>

SELECT ?subject WHERE {?subject geo:lat ?lat; geo:long ?long . FILTER ((xsd:double(?lat) - 37.785834 <= 0.040000) && (37.785834 - xsd:double(?lat) <= 0.040000) &&(xsd:double(?long) - -122.406417 <= 0.040000) && (-122.406417 - xsd:double(?long) <= 0.040000))}

When executing this query, the first step would be to generate the query prefix map. This map would look like this:

- Query prefix → URI: *{geo=http://www.w3.org/2003/01/geo/wgs84_pos#, xsd=http://www.w3.org/2001/XMLSchema#}*

When building Lucene queries based on the filter clauses in the query, it is necessary to first look up the query prefix specified in the query. This query prefix will be used to look up the full URI of the prefix. Finally, the full URI will be used to look up the local prefix in order to communicate with the index through Lucene queries. Based on the example query, the steps of transforming the query prefixes to the local prefixes stored in the index looks like this:

- Predicate → *geo:lat*
- Query prefix → *geo*
- Looking up "geo" in the "query prefix to URI" map*: geo* → *http://www.w3.org/2003/01/geo/wgs84_pos#*
- Looking up the URI in the "full URI to local prefix" map: *http://www.w3.org/2003/01/geo/wgs84_pos#* → *p3*

This process has given us "p3" as the local prefix stored in the index to replicate the predicate "http://www.w3.org/2003/01/geo/wgs84_pos#", represented by the query prefix "geo" in the query. This local prefix must be used when executing Lucene queries based on the filter clauses in the SPARQL query where the predicate "http://www.w3.org/2003/01/geo/wgs84_pos#" is referred to. This merely explains how the conversion between namespaces in terms of querying occurs, but there is one more important step when it comes to transforming namespaces, namely displaying the correct query output to the user. If the Lucene queries based on the SPARQL filter clauses return true, they naturally display results containing the local prefixes instead of the full URIs or the query prefixes, as the local prefixes are stored in the index. This means that the system has to convert the query output to the full URIs in order for the user to understand the results. This is a fairly easy process as the only step needed to transform the query output is to look up the local prefix in the "local prefix to full URI" map that was generated in the indexing process. This means that every data resource of the query output, represented by a URI containing a local prefix, can easily be transformed to the full URI by looking up the local prefix and replacing it with the full URI returned from the map.

3.3 Executing SPARQL filter clauses through Lucene

FILT translates SPARQL queries into Lucene queries in order to retrieve information from the pre-stored index. Only SPARQL queries with filter clauses run through the index. All other queries either run through the local model or the SPARQL endpoint specified by the data set owner. This section will describe in detail how the querying module of FILT works, including the building of Lucene queries based on SPARQL filter clauses, the different types of queries being constructed, and how the query output of these queries is put into an internal data model with the purpose of executing the final SPARQL query stripped of filter clauses.

3.3.1 Filter clauses and corresponding Lucene queries

The SPARQL query language contains numerous filter clauses that allow execution of certain operations on the SPARQL query variables (Prud'hommeaux & Seaborne, 2008). This section will describe in detail the SPARQL filter clauses that have been implemented in FILT and how FILT deals with these filter clauses. The definition of every SPARQL filter clause presented in this section can be found in section 2.1.2.1.

Due to the limited time frame of this project, not all existing SPARQL filter clauses could be implemented in FILT. The filter clauses that have been implemented in FILT are:

- isIRI
- isLiteral
- str
- lang
- datatype
- logical expression
- regex

3.3.1.1 regex, str & lang

The "regex", "str" and "lang" filter clauses are filtered in similar ways through FILT. This section will elaborate on how FILT executes the given filter clauses.

An example of regex filtering in SPARQL can look like this:

*SELECT * WHERE {?s rdfs:label ?label. Filter regex(?label, "This is a regex filter value")}*

In FILT, the regex filter clause is executed through the RegexQuery class in Lucene. This query class allows regular expression to be matched against text stored in the index documents. The default regex compiler used by the RegexQuery class is based on matching the given regular expression input against the entire strings index. This means that it does not match if a regular expression occurs in the text strings, but rather if the regular expression matches the entire text strings. Therefore, FILT overrides the built-in regex implementation of Lucene by implementing a regex compiler that matches regular expressions as an expression in a specific string, rather than an expression as the entire string.

In FILT, the “str” filter clause is executed through the PhraseQuery (Apache Lucene API, 2012) class in Lucene. The “lang” filter clause is queried through the RegexQuery or PhraseQuery class, depending on if other filter clauses are applied to the variable filtered through the “lang” filter clause. For instance, if a “regex” filter clause is applied to the variable also filtered through the “lang” filter clause, the “lang” filter clause will run as a RegexQuery. However, if a “str” filter clause is applied to the variable also filtered through the “lang” filter clause, the “lang” filter clause will run as a PhraseQuery. To elaborate on this, the structure of language literals in the FILT index, mentioned in section 3.1.1.3, must be taken into consideration. Since FILT stores each literal with different languages in fields with different names, such as rdfs:label-en for an object with the English language, rdfs:label-no for an object with a Norwegian language, and so on, the language filtering is simply applied by adding the language tag at the end of the document field name. To demonstrate this, have a look at the query:

SELECT ?label WHERE {?s rdfs:label ?label. Filter (lang(?label) = ‘en’)}

Based on this filter clause, the RegexQuery in Lucene query would look like this:

rdfs:label-en:.

The dot, “.”, in the regular expressions represents “any character”, meaning that the query merely checks that the English language document field exists in the index. If the filter clause had a string value filtering as well, represented by the “str” filter clause, the query would be different. To show this, we can add another filter clause in the query:

SELECT ?label WHERE {?s rdfs:label ?label. Filter (lang(?label) = ‘en’). Filter (str(?label) = ‘Norway’)}

This query would result in the following Lucene PhraseQuery query:

rdfs:label-en:”Norway”

Instead of simply checking that the language document field exists in the index, which the previous query demanded, the query now additionally specifies a value, in this case “Norway”, that must be present in the given language field. The PhraseQuery is adopted when querying the “str” filter clause, instead of the RegexQuery, as the PhraseQuery matches an entire string for the given value, opposed to the RegexQuery, which matches regular expressions in the string. Moreover, the whole expression means that the text value must match “Norway”, and only “Norway”, as the principle of filtering with the “str” is to match an entire value, not parts of a value that the “regex” filter clause allows. However, if a “regex” filter clause was applied to the “?label” variable instead of the “str” filter clause, the PhraseQuery would be replaced by a RegexQuery. To illustrate this, imagine the SPARQL query looking like this instead:

SELECT ?label WHERE {?s rdfs:label ?label. Filter (lang(?label) = 'en'). Filter regex (?label, 'Norway')}

This query would result in the following Lucene RegexQuery query:

rdfs:label-en:Norway

This query looks exactly the same as the PhraseQuery, but is of the type RegexQuery and will match "Norway" in the rdfs:label-en field as an expression or term, instead of a full phrase. This means that a value containing "The Kingdom of Norway" would return true, whereas with the "str" filter clause the same value would be false, as the "str" filter clause has to match the string as a whole.

3.3.1.2 logical expressions

This section will address the term "logical expression" to include filter clauses that have not been assigned any specific filter clause operator. Moreover, filter clauses that are of the syntax "Filter (<filter value>)" will be referred to as "logical expression" filter clauses. The "logical expression" filter clause can filter constraints between variables of a graph pattern, numerical values, and group other filter clauses together. This section will elaborate on how FILT filters numbers, as non-number filtering in the "logical expression" filter clause is merely transformed into queries of the type Lucene RegexQuery. To illustrate the number filtering of FILT, look at this SPARQL query containing a "logical expression" filter clause for filter numbers:

*SELECT * WHERE {?s geo:lat ?lat; geo:long ?long. Filter(xsd:double(?lat) > 50 && ?long = 60)}*

This objective of this filter clause is to find all data entities where the latitude is above 50 and the longitude equals 60. These expressions can easily be translated into existing Lucene queries, namely the NumericRangeQuery and the RegexQuery classes. The first expression "xsd:double(?lat) > 50" is translated into the NumericRangeQuery "geo:lat:[50 TO *]" and the second expression "?long = 60" is transformed into the RegexQuery "geo:long:60". In this case, the NumericRangeQuery "geo:lat:[50 TO *]" has defined the lower term in the query to be exclusive, meaning that only data entities with a latitude over 50 returns true. If the lower term was set to be inclusive, data entities with a latitude equaling 50 would also return true. This would be correct to apply if the filter expression rather stated "xsd:double(?lat) >= 50". The same principles apply to any NumericRangeQuery, whether the query contain only a lower term or an upper term, or both. Any expression containing the EQUAL expression operator ("=") or the NOT EQUAL expression operator ("!="), regardless of filter value, is translated into the RegexQuery. If the query is based on the equal operator, it will only include the filter value itself as the query input, such as the query just mentioned: "geo:long:60". However, if the filter expression stated "?long != 60" instead of "?long = 60", the RegexQuery would have to generate a regular expression with a “negative look ahead” condition, in order to find data entities with a latitude not matching the value “60”. This RegexQuery would look like this:

geo:long^(?!.*60).*$)

See Table 3.4 for the different expression operators and their related Lucene queries.

Table 3.4: Expression operators and their related Lucene queries

Expression operators	Lucene query
=	RegexQuery
!=	RegexQuery
>	NumericRangeQuery (exclusive term)
<	NumericRangeQuery (exclusive term)
>=	NumericRangeQuery (inclusive term)
<=	NumericRangeQuery (inclusive term)

The built-in Lucene query library offers the possibility of easily translating simple number filtering into different queries. However, more complex number filtering cannot be directly translated into Lucene queries. This can be demonstrated through this query:

SELECT ?subject WHERE {?subject geo:lat ?lat; geo:long ?long . FILTER ((xsd:double(?lat) - 37.785834 <= 0.040000) && (37.785834 - xsd:double(?lat) <= 0.040000) &&(xsd:double(?long) - -122.406417 <= 0.040000) && (-122.406417 - xsd:double(?long) <= 0.040000))}

The filter clause expressions in this query is tricky to filter by using Lucene queries, as none of the built-in Lucene query classes can execute mathematical expressions containing the numeric operators “addition”, “subtraction”, “division” and “multiplication”. This means that in order to execute the number filtering expressions in the filter clause, the mathematical expressions have to be simplified in order to meet the requirements of the Lucene query libraries. FILT translates complex numeric expressions into more simple expressions in order to meet the requirements of the built-in Lucene query library. The rules for simplifying the numerical expressions are based on the standard mathematical rules for equations and inequalities. The entire set of rules for converting the numerical expressions are presented in Appendix 1. This text will present one example for converting the numerical expressions. The example looks like this:

If the numerical operator is subtraction:

Rule: subtract the same number on both sides and reverse the orientation of the inequality sign

Example:

(37.785834 - xsd:double(?lat) <= 0.040000):

37.785834 – 0.040000 - xsd:double(?lat) >= 0.040000 - 0.040000

37.745834 - xsd:double(?lat) >= 0

(xsd:double(?lat) >= 37.745834)

?lat must equal or have a higher value than 37.745834 in order for the initial expression to be true

To demonstrate an excerpt of the rules for simplifying numerical expression in Appendix 1, have a look at the query previously described:

SELECT ?subject WHERE {?subject geo:lat ?lat; geo:long ?long . FILTER ((xsd:double(?lat) - 37.785834 <= 0.040000) && (37.785834 - xsd:double(?lat) <= 0.040000) && (xsd:double(?long) - -122.406417 <= 0.040000) && (-122.406417 - xsd:double(?long) <= 0.040000))}

The “logical expression” filter clause numerical expressions in this query are:

((xsd:double(?lat) - 37.785834 <= 0.040000) && (37.785834 - xsd:double(?lat) <= 0.040000) && (xsd:double(?long) - -122.406417 <= 0.040000) && (-122.406417 - xsd:double(?long) <= 0.040000))

Based on the mathematical rules for inequalities, the initial expressions are simplified into the following expressions:

((xsd:double(?lat) <= 37.825834) && (xsd:double(?lat) >= 37.745834) && (xsd:double(?long) <= -122.366417) && (xsd:double(?long) >= -122.44641700000001))

3.3.1.3 isIRI & isLiteral

FILT executes “isIRI” and “isLiteral” filter clauses through the built-in method “isURI” in the Jena framework. In order to this, it is necessary to obtain the value of the variable specified in the filter clause. This is done by retrieving a random value stored in the index field specified by the variable in the filter clause. The “isURI” method is a Boolean check and returns true if the value retrieved from the index is a URI, and false otherwise. The value given as a parameter to the method must be of the data type string. If the “isURI” method returns true for a value in an “isIRI” filter clause, the filter clause is true. If the method returns false, the filter clause is also false. If the method returns true for a value in an “isLiteral” filter clause, the filter clause is false. If it returns true, the filter clause is false. To illustrate the execution of “isIRI” and “isLiteral” filter clauses in FILT, have a look at the following SPARQL query:

SELECT ?person WHERE {?subject foaf:knows ?person. Filter (isIRI(?person))}

As the foaf:knows predicate restricts the subject and the object of the triple to be of a type “foaf:Person”, this means that the variable ?person should be represented by a URI. Moreover, the “isIRI” filter clause should return true. In order to check this, the values of the objects in the triple is sent as a parameter value to the “isURI” method, and the method returns true if the value is a URI, and false if otherwise. The same principle is adopted when executing “isLiteral” filter clauses.

3.3.1.4 datatype

In order to understand how FILT deals with “datatype” filter clauses, it is necessary to have a look at how FILT index data type metadata. As mentioned in section 3.1.1.3, FILT stores data type metadata as separate fields in the index document being stored. The data type document fields are related to the predicate they represent, meaning that the data type index fields themselves have dynamic values based on what predicate they correspond to. For example, the latitude object value of a triple containing the geo:lat (http://www.w3.org/2003/01/geo/wgs84_pos#lat) predicate looks like this: "50.10"^^<http://www.w3.org/2001/XMLSchema#double>. The data type metadata of this value would be stored in a separate field with the name “geo:lat-datatype” and the value “http://www.w3.org/2001/XMLSchema#double”. In order to illustrate how “datatype” filter clauses execute through FILT, have a look at the following SPARQL query:

SELECT ?subject WHERE {?subject geo:lat ?latitude. Filter(datatype(?latitude) = xsd:double)}

The value of the “datatype” filter clause is “xsd:double”. This value is converted into its full URI through the namespace maps mentioned in section 3.2.3, in this case “http://www.w3.org/2001/XMLSchema#double”. Similar to the “str” filter clauses, the “datatype” filter clauses are executed through the Lucene PhraseQuery. The PhraseQuery matches a specific phrase to the entire value of a document field. As the data type fields stored in the index only contains the data type URI itself, this means that a PhraseQuery with a mere URI input is exactly what is needed to match the value of a data type document field. The PhraseQuery based on the SPARQL query example would look like this:

geo:lat-datatype:"http://www.w3.org/2001/XMLSchema#double"

This query will match the entire string “http://www.w3.org/2001/XMLSchema#double” in the data type field of the geo:lat predicate, which has a stored value of “http://www.w3.org/2001/XMLSchema#double”. Hence, the PhraseQuery would return true.

3.3.2 Building and executing Lucene queries based on SPARQL filter clauses

The Lucene queries are built based on the filter clauses in the given SPARQL query that is being executed, and each specific filter clause is converted to one or more separate Lucene queries. When every filter clause has been divided into distinct Lucene queries, these different Lucene queries will be joined as one large query and finally executed over the index. To illustrate this, have a look at this SPARQL query example:

SELECT ?s WHERE {?s rdf:type ?type; geo:lat ?lat; geo:long ?long. Filter regex(?type, “Location”). Filter (xsd:double(?lat) > 50 && geo:long > 10)}

This query contains two different filter clauses:

- *Filter regex(?type, “Location”)*
- *Filter (xsd:double(?lat) > 50 && xsd:double(?long) < 10)*

The filter clauses are put into a “filterClauseMap” in order to efficiently keep hold of the predicate that is being filtered, the type of the filter clause, and the filter value. The “filterClauseMap” is analyzed by the query module and queries are built based on its content. It is important to note that all logical expressions are being split into separate queries every time an AND (“&&”) or OR (“||”) operator occurs. This is due to the fact that these expressions can contain different predicates, meaning that different fields in the index have to be looked up, thus making it necessary to split the expressions into separate queries before eventually joining them as one final query. To illustrate this aspect, the filter clauses in the SPARQL query will be translated into these Lucene queries:

- rdf:type:Location
- geo:lat:[50 TO *]
- geo:long:[* TO 10]

This example illustrates how the second filter clause in the query is divided into two queries, given the fact that the filter value contains two logical expressions separated by the AND operator. The logical operators do not merely serve the purpose of defining when to split the filter values into separate Lucene queries, they also play an important role when it comes to the querying itself. The logical operators determine if an expression must occur or should occur in the query. The AND operator implies that the expressions on both sides of the operators have to return true in order for the expression as a whole to be true. However, the OR operator infers that only *one* of the logical expressions on each side of the operator must return true in order for the entire expression to be true. Moreover, it is not sufficient to build Lucene queries based on the logical expressions isolated from one another; the logical operators connecting these expressions have to be taken into consideration in terms of combining the queries together. The system applies logical operators to the queries by mapping each query to its logical operator, represented by the BooleanClause.Occur class in Lucene. The BooleanClause.Occur class keeps hold of the occurrence of a given query and can be set to three different conditions:

- MUST
- MUST NOT
- SHOULD

The MUST condition (represented by a "+" character in Lucene queries) indicates that the value of a given must occur in the index for it to return true, the MUST NOT condition (represented by a "-" character in Lucene queries) indicates that the value of a given query must not occur in the index for it to return true, and the SHOULD condition (represented by a blank character, "", in Lucene queries) indicates that the value of a given query *should* occur, but does not have to occur in the index in order for it be true. In FILT, the MUST and SHOULD conditions are used to represent the AND and OR logical operators in SPARQL queries. Each separate Lucene query being built in FILT is mapped to a BooleanClause.Occur reference based on the logical operator that connects the logical expressions to other expressions. If two expressions are divided by the AND operator, both of the queries constructed from the two expressions will be assigned the BooleanClause.Occur.MUST condition, due to the fact that both of the expressions have to be true in order for the entire logical expression to return true. If two logical expressions are divided by the OR operator, both of the queries constructed based on the expressions will be given the BooleanClause.Occur.SHOULD condition, meaning that one of the expressions should occur in the index in order for the query to return true. Filter clauses that only consists of one expression, or value, are assigned the BooleanClause.Occur.MUST condition, meaning that it must occur in the index in order to be true. Based on the previous example of Lucene queries constructed from the SPARQL query, the final mapping of the queries and their occurrence-conditions will look like this:

- rdf:type:Location → BooleanClause.Occur.MUST
- geo:lat:[50 TO *] → BooleanClause.Occur.MUST
- geo:long:[* TO 10] → BooleanClause.Occur.MUST

Note that the “rdf:type:Location” query is assigned the BooleanClause.Occur.MUST condition, simply because by being expressed in the SPARQL query it is implied that it must occur in order for the query as a whole to return true. If the second filter clause in the SPARQL query (“Filter (xsd:double(?lat) > 50 && xsd:double(?long) < 10)”) was to use the OR operator instead of the AND operator to connect the two expressions, the queries and their occurrence mapping would look like this:

- rdf:type:Location → BooleanClause.Occur.MUST
- geo:lat:[50 TO *] → BooleanClause.Occur.SHOULD
- geo:long:[* TO 10] → BooleanClause.Occur.SHOULD

Finally, when every filter clause has been analyzed and converted into different Lucene queries with a mapping to BooleanClause.Occur conditions, a final joined query consisting of all the constructed Lucene queries and their occurrence-conditions is constructed. This query is constructed through the BooleanQuery class in Lucene, and the final query construction based on the previously described query mapping looks like this:

```
BooleanQuery finalQuery = new BooleanQuery();
BooleanQuery shouldOccurQuery = new BooleanQuery();
int queryNumber = 1;
while(finalQueryMap.containsKey(queryNumber)) {

      Iterator<Query> it = finalQueryMap.get(queryNumber).keySet().iterator();
      while (it.hasNext()) {
            Query query = it.next();
            Occur booleanClause = finalQueryMap.get(queryNumber).get(query);
            if (booleanClause.equals(BooleanClause.Occur.MUST)) {
               if (!shouldOccurQuery.equals(new BooleanQuery())) {
                   finalQuery.add(shouldOccurQuery, booleanClause);
                   shouldOccurQuery = new BooleanQuery();
                   }
             finalQuery.add(query, booleanClause);
             }
             else {
                  shouldOccurQuery.add(query, booleanClause);

             }
        }
        queryNumber++;
}
```

This loop iterates through the “finalQueryMap” which has a data structure of HashMap<Integer, HashMap<Query, Occur>>. The Integer key of this map is signified by a query number, representing the position of the filter clause expression, or value, in the SPARQL query. A Lucene query based on a filter clause expression that occurs first in the SPARQL query is given the query number “1”, and the query based on the second filter clause expression in the query is given the query number “2”, and so on. This Integer key is vital for knowing what queries that should be combined with each other. To illustrate this, take into consideration the second filter clause of the SPARQL query described earlier in this section: “Filter (xsd:double(?lat) > 50 && xsd:double(?long) < 10)”. The occurrence-condition that eventually will combine the two expressions on each side of the AND operator is the BooleanClause.Occur.MUST. In order to make sure that the Lucene queries based on the two logical expressions on each side of the AND operator are combined with each other, and not any other queries, we have to make sure they are put in an order next to each other. Henceforth, they have to be assigned chronologically query numbers, for instance “2” and “3”.

The value of the “finalQueryMap” is a HashMap with a key of the type “Query” and a value of the type “Occur” (“BooleanClause.Occur”). This map contains all the constructed Lucene queries based on the filter clause expressions in a given SPARQL query, along with the occurrence-conditions of these queries. If a query has an assigned occurrence-condition of the type BooleanClause.Occur.SHOULD, the query is added to the BooleanQuery “shouldOccurQuery”. The next query, or queries, with an assigned BooleanClause.Occur.SHOULD occurrence-condition will further be added to the same “shouldOccurQuery” before eventually adding the joined query to the superior query “finalQuery”. This serves the purpose of creating one large query combining every query constructed from logical expressions that are joined with the OR operator. If these queries were not combined together before being added to the “finalQuery”, the final query syntax would be incorrect. This is due to the fact that the BooleanClause.Occur.SHOULD is describing the relationship between the queries constructed from the logical expressions on each side of an OR operator. However, adding them directly into the final query would mess up the relationship to the other queries already added. The “shouldOccurQuery” still has to be added to the final query with the BooleanClause.Occur.MUST condition, because the joined query itself has to return true in order for the entire “finalQuery” to be true. To illustrate this, imagine the second filter clause of the SPARQL query mentioned previously in this section looking like this: “Filter (xsd:double(?lat) > 50 || xsd:double(?long) < 10)”. This would be translated into these Lucene queries and occurrence-mappings:

- geo:lat:[50 TO *] → BooleanClause.Occur.SHOULD
- geo:long:[* TO 10] → BooleanClause.Occur.SHOULD

These queries would accordingly be added to the “shouldOccurQuery” like this:

*(geo:lat:[50 TO *] geo:long:[* TO 10])*

Note that the queries are separated by a blank character, or a whitespace. As mentioned earlier in this section, this character signifies the BooleanClause.Occur.SHOULD condition in Lucene queries. Finally, the “shouldOccurQuery” is added to the “finalQuery” by applying the BooleanClause.Occur.MUST condition (this example infers that the query constructed from the first filter clause in the SPARQL query has already been added to the “finalQuery”):

(+rdf:type:Location +*(geo:lat:[50 TO *] geo:long:[* TO 10]))*

The BooleanClause.Occur.MUST condition is represented by the “+” character, as mentioned previously in this chapter. In this example, the “finalQuery” contains two queries, in this case the rdf:type query and the “shouldOccurQuery”, which MUST occur in the index to return true. If the “shouldOccurQuery” was not added to the “finalQuery” with the BooleanClause.Occur.MUST condition, but rather the BooleanClause.Occur.SHOULD condition, the query would instead state that the “shouldOccurQuery” SHOULD occur in the index, meaning that if the “shouldOccurQuery” did not return true in the index, the query would still be true, as long as the rdf:type query returned true. This would be incorrect, as the SPARQL query clearly states that both filter clauses must be true in order for the query to return true. Figure 3.2 shows the query building architecture of FILT.

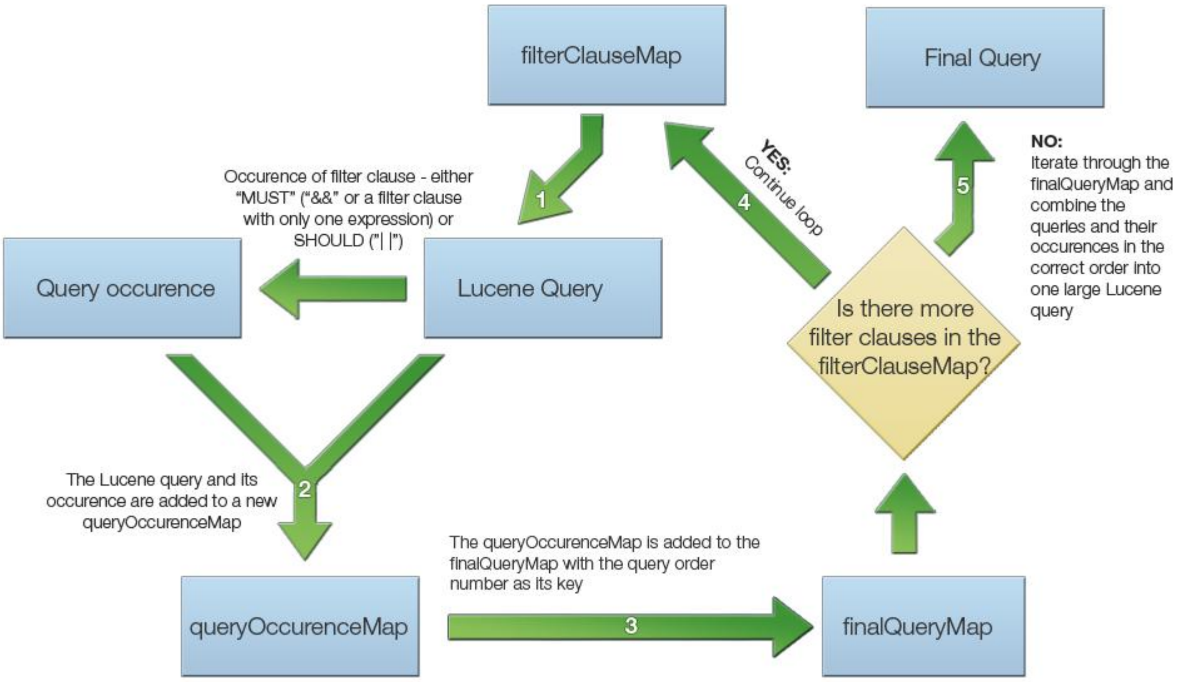

filterClauseMap
Final Query
YES:
Continue loop
4
NO:
Iterate through the finalQueryMap and combine the queries and their occurences in the correct order into one large Lucene query
5
Occurence of filter clause - either "MUST" ("&&" or a filter clause with only one expression) or SHOULD ("| |")
1
Query occurence
Lucene Query
Is there more filter clauses in the filterClauseMap?
The Lucene query and its occurence are added to a new queryOccurenceMap
2
The queryOccurenceMap is added to the finalQueryMap with the query order number as its key
3
queryOccurenceMap
finalQueryMap

Figure 3.2: The query building architecture of FILT

3.3.3 Constructing an RDF model based on the Lucene query output

The Lucene queries based on the SPARQL filter clauses quickly retrieves the full set of data which are necessary for executing the general SPARQL query without filter clauses. However, the Lucene queries do not take into consideration the remaining general SPARQL query, stripped of the filter clauses. The general SPARQL query is executed at a later stage when the Lucene queries have retrieved data corresponding only to the filter clauses in the SPARQL query. The data retrieved by the Lucene queries is not the correct query output of the entire SPARQL query, merely the filter clauses of the SPARQL query. The Lucene queries obtain the relevant triples matching the filter clauses in the SPARQL query, and these triples are the basis for a new RDF model where the general SPARQL query will be executed in order to find the correct results of the entire SPARQL query. Moreover, the SPARQL query as a whole is executed in two steps: Lucene queries constructed from the filter clauses, and a general SPARQL query executed over a temporary local Jena RDF model constructed from the triples matching the filter clauses from the SPARQL query. If the Lucene queries correspond to one or more data entries in the pre-stored index, the queries will return true, and the system selects the relevant triples needed to build the local RDF model. These triples are selected based on the predicates defined in the general SPARQL query, as the general SPARQL query has the purpose of making sure that the data entities corresponding to the Lucene queries contain the triples stated in the SPARQL query. This can be illustrated by having a look at the same SPARQL query example from section 3.3.2:

SELECT ?s WHERE {?s rdf:type ?type; geo:lat ?lat; geo:long ?long. Filter regex(?type, "Location"). Filter (xsd:double(?lat) > 50 && geo:long > 10)}

As described in section 3.3.2 the final Lucene query based on the filter clauses in the SPARQL query will look like this:

*(+rdf:type:Location +(geo:lat:[50 TO *] geo:long:[* TO 10]))*

This Lucene query will find all the data entities in the pre-stored index that correspond to the SPARQL filter clauses, but it does not provide any information regarding to what specific query output to retrieve from the query. In order to know what data to retrieve from the relevant data entities the general SPARQL query has to be generated. The general SPARQL query from the same example looks like this:

SELECT ?s WHERE {?s rdf:type ?type; geo:lat ?lat; geo:long ?long.}

The general query offers all the information needed in order to know the conditions the data entities have to fulfill in order to be true for the entire SPARQL query. To elaborate, the general SPARQL query contains information regarding what triples that need to be retrieved from the index in order to build the local RDF model for further querying. There is no need for fetching triples that are not defined in the SPARQL query, as these triples are irrelevant and would only consume more resources, thus increasing the query execution time. The general SPARQL query is being executed over the generated RDF model in order to answer the entire SPARQL query, not just the filter clauses in the query. Based on the general SPARQL query example the triples that must be present in the RDF graph of the entities are:

- *?s rdf:type ?type*
- *?s geo:lat ?lat*
- *?s geo:long ?long*

This means that for each result the Lucene queries give in the index, the system has to retrieve the subject URI of the RDF graph, the rdf:type predicate and its object, the geo:lat predicate and its object, and the geo:long predicate and its object. If the RDF graph contains more triples than the ones corresponding to the graph patterns defined in the SPARQL general query, these triples are being ignored, as they are not necessary to take into consideration in order to answer the general SPARQL query. The system knows that these data values have to be extracted from the index by analyzing the general SPARQL query prior to executing the Lucene queries. The triples retrieved from the index will be constructed as an RDF model where the general SPARQL query will be executed. If the sole purpose of executing the general SPARQL query was to check if the RDF graphs of the different data entities contained the triples defined in the general SPARQL query, it would not be necessary to execute the general SPARQL query at all, as this check is already done when the relevant triples are being retrieved from the index. However, SPARQL queries can contain certain operators such as “LIMIT”, which specifies a limit to the query, and “ORDER BY”, which determines the order of the result set sequence. Such operators are best executed through the SPARQL query language, thus making it necessary to execute the general SPARQL query over the generated RDF model based on the triples retrieved from the index.

Chapter 4: Methodology

This section will describe the research methodology adopted in this project, the system development research framework implemented when developing FILT, and the framework for evaluating FILT through an extensive benchmark test.

4.1 Design research

The research reported in this thesis is an example of design research. Design research aims at creating innovative artifacts consisting of new ideas, practices and technical possibilities. These artifacts should act as the basis for the data collection, analysis and evaluation of the project (Denning 1997; Tsichritzis 1998, cited by Hevner, March, Park & Ram, 2004).

This project has used the framework for research processes of system development research methodologies, presented by Nunamaker Jr. and Chen (1990) (see Figure 4.1). The framework emphasize the importance of doing continually evaluations of the system through several iterations, which is something that was done by indexing data and running test queries throughout the entire system development cycle.

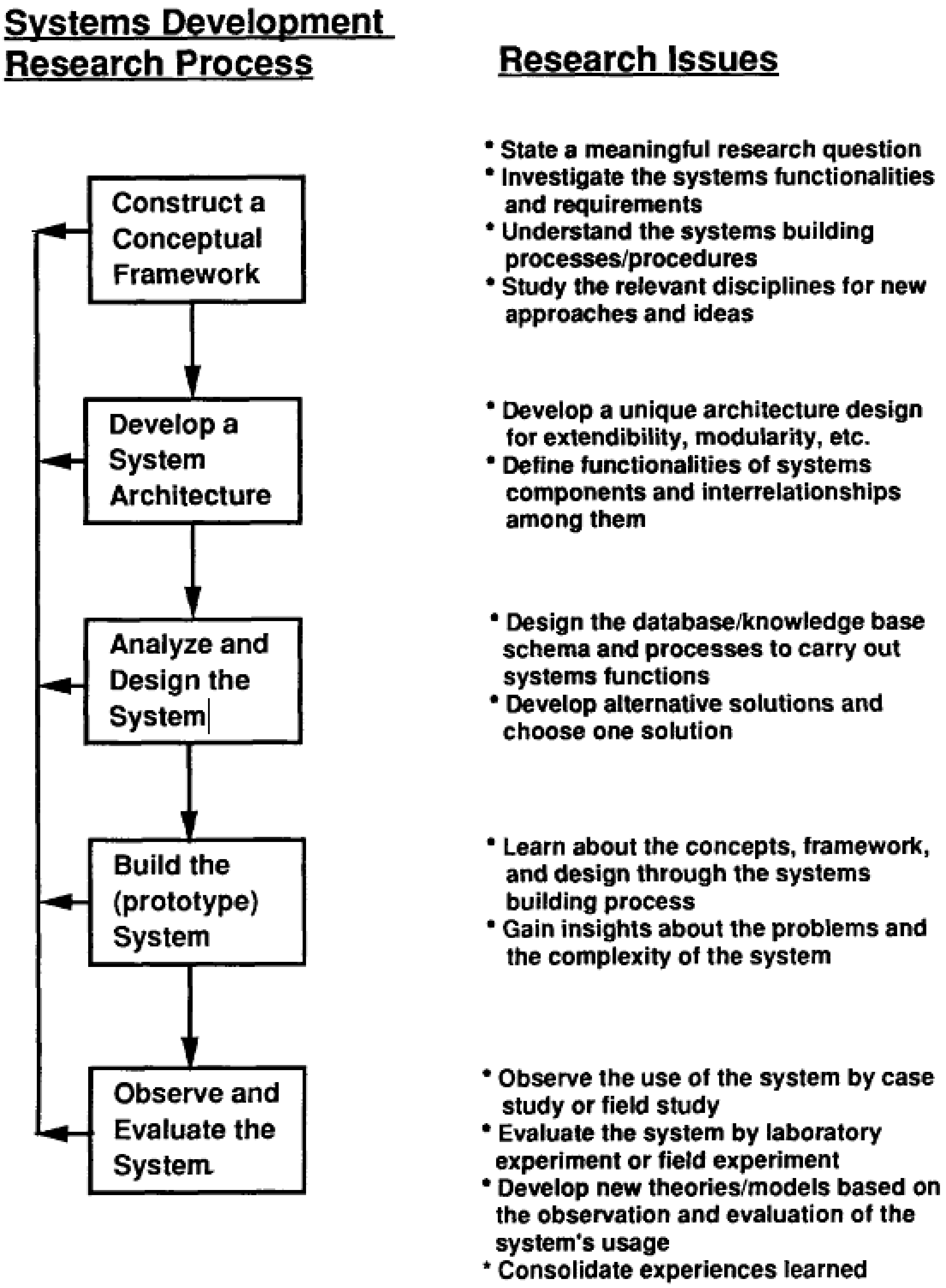

Systems Development Research Process
Research Issues
Construct a Conceptual Framework
• State a meaningful research question
• Investigate the systems functionalities and requirements
• Understand the systems building processes/procedures
• Study the relevant disciplines for new approaches and ideas
Develop a System Architecture
• Develop a unique architecture design for extendibility, modularity, etc.
• Define functionalities of systems components and interrelationships among them
Analyze and Design the System
• Design the database/knowledge base schema and processes to carry out systems functions
• Develop alternative solutions and choose one solution
Build the (prototype) System
• Learn about the concepts, framework, and design through the systems building process
• Gain insights about the problems and the complexity of the system
Observe and Evaluate the System
• Observe the use of the system by case study or field study
• Evaluate the system by laboratory experiment or field experiment
• Develop new theories/models based on the observation and evaluation of the system's usage
* Consolidate experiences learned

Figure 4.1: A Research Process of Systems Development Research Methodology (Nunamaker Jr. & Chen, 1990)

The first step in the framework is to construct a conceptual framework. This includes stating a meaningful research question, investigating the functionalities and requirements of the system, understanding the building processes of the system, and study relevant disciplines for new ideas. There were structured four research questions that the project intended to answer, along with three success criteria which would be the basis for the formal system requirements so that the system could contribute to answer the research questions. This phase of the system development cycle was mainly spent reviewing existing literature and relevant work in order to get a full understanding of what others had tried to accomplish before, and learn previous research projects. Also, reading about the Apache Lucene framework and understanding its functionalities were an important aspect in this phase.

The second and third steps in the framework are to a certain extent related to each other, and will be presented together. The second step is to develop the system architecture. This includes building a unique architecture design, and defining the functionalities of the system components and the relationship between them. The third step includes analyzing and designing the system, thus designing the database/knowledge base schema and processes to carry out the system functions, as well as developing alternative solutions and finally choosing one of them. This project spent a great deal of time designing the architecture and how the different system components should interact with each other. When designing a hybrid architecture consisting of a full-text search module and a conventional database retrieval engine, there are many aspects to take into consideration. This project came up with two possible hybrid architecture designs. The first architecture design was constructed based on the idea of the final output of a SPARQL query executed through the system would be an intersection of the results from the general SPARQL query without filter clauses executed through a conventional triple store, and the results returned from the Lucene queries constructed from the filter clauses in the query. Moreover, the final query output would be an intersection between all the results retrieved from the conventional triplestore by executing the general SPARQL query, and the results retrieved from the Lucene queries constructed from the filter clauses in the SPARQL query. The second architecture would not execute the general SPARQL query and the Lucene queries constructed from the filter clauses in the SPARQL query at the same time, but rather execute the Lucene queries before the general SPARQL query. The results from the Lucene queries would be returned as triples, loaded into a local RDF model, and the general SPARQL query would be executed through the local model. This was the final and current implementation of FILT. The three different index structures presented in section 3.1.1 were also designed during the second and third steps of the system development cycle, where the third index structure implementation was decided to be the most suitable to fulfill the research questions and success criteria of the project.

The fourth step in the framework is to build the prototype system by learning about the concepts and frameworks. Gaining insights and learning about the challenges, problems and complexity of the system is an important aspect in this step. During the implementation of FILT several important aspects regarding the storing- and querying mechanisms of the Apache Lucene framework were discovered. The learning process of fully understanding the Apache Lucene framework was a vital component in this project, and was mainly a part of the actual implementation of FILT. It is hard to learn a technical information retrieval framework without actually implementing it and understanding its functionality. Moreover, the implementation phase of FILT included several important findings of how the Apache Lucene framework could supplement the research questions of this project to the fullest. This led to several different implementations and changes being performed continuously during the development cycle. The most important aspect of the implementation phase was the development of the two different query architectures, as mentioned previously. The first architecture including intersecting the results between the general SPARQL query executed in a conventional triplestore, and the results from the Lucene queries, was found to execute queries slowly as general SPARQL queries are often too general and will return enormous amounts of results by executing them through the entire data set. This querying architecture was implemented in FILT at an early stage of the development process, but was discarded throughout the system development cycle due to its slow query-execution times and rigidity. A more flexible architecture was simultaneously developed, namely the architecture of first retrieving relevant RDF triples from the Lucene queries constructed from the SPARQL query filter clauses, and then executing the general SPARQL query over the RDF triples loaded into a local RDF model. This architecture would eventually be the final architecture of FILT and is described in detail in Chapter 3.

The fifth step includes observing and evaluating the system. Observations can be done by performing case studies or field studies, and evaluations can be done by a laboratory experiment or field experiment. The observations and evaluations will be the foundation for developing new theories and models. When this step is finished, the previous steps should be revisited in order to optimize the system based on the experiences and knowledge obtained throughout the development cycle. As FILT is not a front-end system built for direct user-interaction, but rather a back-end database solution, no observations of the system that included end-users took place. However, during the system development cycle, the system was the target of various informal benchmark tests, where several queries were executed and their query-execution times recorded. The query-execution times were compared to the execution times of the same queries executed through conventional triplestores. Informal benchmark tests were performed several times during the development of the system, and these tests indicated that FILT performed faster and faster compared to conventional triplestores. The informal benchmark tests performed during the development of the system provided an indication to till what extent the system currently could fulfill the research questions and success criteria constructed in the initial phases of the project, and was therefore an important aspect in the process of continuously adapting and improving the system through several iterations.

4.2 Benchmark evaluation

In this project, an extensive benchmark evaluation of FILT was performed. The benchmark compared the features of FILT to a conventional triplestore by evaluating several metrics regarding the speed of query execution. This section will clarify certain terms and their definitions, describe the hardware and software used in the benchmark, define the data sets and queries applied, present the metrics used for the performance evaluation of the systems as well as the metrics for designing the queries executed in the benchmark, present a set of rules the benchmark evaluation cycle will follow, and finally describe a framework for presenting the results of the benchmark evaluation.

4.2.1 Definitions of terms

This text will refer to the term “System Under Test (SUT)” in every case where it mentions the system that is currently being tested in the benchmark evaluation. The full specification and definition of SUT is elaborated in (Transaction Processing Performance Council (TCP), 2010:80). Another term that will be referred to in the benchmark evaluation is the “test driver”. The test driver will in this context be defined as an external Driver System that provides Remote Terminal Emulator (RTE) functionality. The RTE must be used to emulate the target terminal population and their emulated users during the benchmark run (Transaction Processing Performance Council (TCP), 2010:80). In this project, the RTE includes the following features:

- Replicating a scenario of a user entering input data by sending transactional requests to the SUT
- Replicating a terminal presenting output by retrieving response messages from the SUT
- Storing of response times

4.2.2 Hardware and software

The following hardware and software specifications for the benchmark evaluation are as following:

- Hardware specifications
  - Intel Core 2 Dual core processor T6400 (2 GHz, 800 MHz FSB, 2 MB L2 cache)
  - 4 GB DDR2 RAM
  - 320 GB Hard drive
  - 802.11a/b/g/Draft N WLAN adapter
  - Windows 7 Ultimate Edition SP 1 operating system
- Test driver

  - Eclipse Java EE IDE for Web Developers. Version: Indigo Service Release 2 Build ID: 20120216-1857. Using the Java Virtual Machine (JVM) version 1.6, update 29. Download location: http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/indigo/SR2/eclipse-jee-indigo-SR2-win32-x86_64.zip
- Tools
  - The Java programming language for executing query execution algorithms
  - The Jena framework for Java for connecting to the SUT

4.2.3 Data sets and queries

The benchmark evaluation included executing two pre-defined sets of SPARQL filter queries over two separate data sets. The two different data sets that the queries were executed over were the DrugBank data set and the Geographic Coordinates RDF graph of the DBpedia data set. The DrugBank data set contains 766,920 triples, whereas the Geographic Coordinates data set contains 1,771,100 triples (see Table 4.1). For this benchmark evaluation, both the DrugBank data set and the Geographical Coordinates (DBpedia) data sets were divided into three data sets; each with a distinct amount of triples. The data sets were split into one sub-set containing 1/7 of the total amount of triples and one sub-set containing 1/2 of the total amount of triples. Finally, the entire data set were tested. Based on this, the DrugBank data set were divided into the following three data sets:

1. a sub-set containing 100,000 triples of the total data set (1/7 of the entire data set)
2. a sub-set containing 350,000 triples of the total data set (1/2 of the entire data set)
3. the entire data set containing 766,920 triples

The Geographic Coordinates (DBpedia) data set were divided into the following three data sets:

1. a sub-set containing 253,000 triples (1/7 of the entire data set)
2. a sub-set containing 885,000 triples of the total data set (1/2 of the entire data set)
3. the entire data set containing 1,771,100 triples

These data sets were loaded into two different data stores: FILT and Joseki. Joseki is a triplestore for Jena, developed by W3C RDF Data Access Working Group. It supports the SPARQL protocol and the SPARQL RDF Query Language. The version of FILT that was applied in the benchmark evaluation is v1.0, and the Joseki version used is v3.4.4.

The query mixes were executed over each of the divided data sets, both through the Joseki triplestore and FILT, in order to illustrate the scalability performance of a conventional triplestore opposed to FILT.

The data sets included in the benchmark evaluation were downloaded at the 8th of March, 2012 and can be accessed through the following URLs:

- DrugBank: http://www4.wiwiss.fu-berlin.de/drugbank/drugbank_dump.nt
- Geographical Coordinates of DBpedia:
  http://downloads.dbpedia.org/3.7/en/geo_coordinates_en.nt.bz2

Table 4.1: The data sets implemented in the benchmark evaluation

Data set	Number of triples
DrugBank	766,920
Geographic Coordinates (DBpedia)	1,771,100

The two data sets were chosen based on the two use-cases in section 2.2.2. As presented in both Chapter 1 and previous sections in this chapter, SPARQL filter queries provide multiple possibilities of finding information that could not be found through general SPARQL queries without filter queries. However, the downside of SPARQL filter queries is that these queries generally execute slowly. Instead of simply matching a graph-pattern, which is the case in general SPARQL queries, SPARQL filter queries have to filter through a wide variety of data values stored in the triples. This will naturally lead to slower query execution times opposed to general SPARQL queries. Based on this, this project aims at discovering techniques and principles for optimizing the query-execution times of SPARQL filter queries, and building a prototype solution called FILT to show that the query-execution time of SPARQL queries can be decreased noticeably by implementing the Apache Lucene framework for performing full-text searches and filtering logical/numerical expressions, which act as the foundation for the research questions formed in section 2.2.4. The two use-cases will thus shed light on the two major features of FILT, namely the execution of regex filtering and numerical filtering through SPARQL queries. In order to do this, two separate sets of relevant SPARQL queries have been put together; one set of queries corresponding to each of the DrugBank and Geographical Coordinates data sets, and highlighting each of the two most important features of FILT. Moreover, the queries running through the DrugBank data set will mainly be focused on executing SPARQL filter queries containing the regex filter clause, whereas the queries executed over the Geographical Coordinates data set will predominantly be SPARQL filter queries containing numerical expressions. The metrics for how the queries should be constructed are described in section 4.2.4.2.

4.2.4 Metrics

This section will specify the metrics for the benchmark evaluation itself, as well as metrics for constructing the queries that will be executed in the benchmark evaluation.

4.2.4.1 Performance evaluation metrics

The dependent variable in the benchmark tests is query-execution time. In this evaluation, query-execution time will be defined as *the time spent from executing a query request to a database till the entire result set from the query request has been returned.* This definition supports a real-world scenario of a user issuing a query to a database and retrieving the output of the query. In order to measure the query-execution time, four specific metric variables have been designed. The metrics are based on the performance metrics specified by (Bizer & Schultz, 2009), but differ slightly. The metrics of the benchmark evaluation are "Milliseconds per Query (MSpQ)", "Average Query Execution Time (aQET)", "Overall Runtime (oaRT)" and "Average Query Execution Time over all Queries (aQEToA)". All the metrics and their definitions are shown in Table 4.2.

Table 4.2: The performance metrics and their definitions

Metrics for single queries	**Definition**
Milliseconds per Query (MSpQ)	The amount of milliseconds spent on executing one single query
Average Query Execution Time (aQET)	Average time for executing a single query multiple times
Metrics for query mixes	**Definition**
Overall Runtime (oaRT)	Overall time it takes the test driver to execute a query mix against the SUT
Average Query Execution Time over all Queries (aQEToA)	Overall time to run a query mix divided by the number of queries

The benchmark evaluation will only evaluate and present the aQET. The aQET was calculated by the average time it takes to execute a single query multiple times. The aQET of each query will then be combined with the aQET of the queries of the same query form. Moreover, this means that the aQET of all SELECT queries will be calculated into a combined aQET for SELECT queries. The same procedure will be repeated with all query forms. This way it is possible to analyze the performance of the two data stores based on different query forms.

4.2.4.2 Query metrics

- All queries in both the use-cases shall run with the same pre-defined set of prefixes, which will cover all of the prefixes needed to execute each and all of the queries. This way, possible query-execution time-differences related to prefixes will be eliminated.
- All SPARQL query forms will be implemented in the query mix, including SELECT, DESCRIBE, CONSTRUCT and ASK. The query mixes will consist of six queries of each query form. This way, the benchmark evaluation will provide an indication as to how FILT performs with every query form of the SPARQL query language
- All queries shall consist of basic graph pattern matching and filter clauses, and will thus not include additional pattern matching or complex functions. This is because the main purpose of FILT is to increase query-execution time of SPARQL filter clauses, meaning that any component of a SPARQL query, apart from its filter clauses, will execute in a conventional way. Hence, evaluating the performance of other functions apart from filter clauses will have a small, if any, impact on the test results
- Each query mix shall contain queries with one single filter clause and queries with two or more filter clauses, in order to measure in the query-execution time of queries with a single or multiple filter clauses

Based on the query metrics, the set of prefixes that will be attached to every SPARQL query in the query mixes are:

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dcterms: <http://dublincore.org/2010/10/11/dcterms.rdf#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX dbpedia: <http://dbpedia.org/resource/>

The text will further present the query mixes of each of the different use-cases of the benchmark evaluation.

4.2.5 Query mixes

4.2.5.1 The query mix for the DrugBank data set

This section will present the query mix for the DrugBank data set. Table 4.3 shows the natural language queries, whereas Table 4.4 shows the natural language queries translated into a SPARQL representation.

Table 4.3: The query mix for the DrugBank data set

Queries
1. Find the distinct labels of entities where the textual descriptions contain the regular expressions “Interferon” and “theophylline”
2. Find the distinct labels of entities where the indication/treatment descriptions contain the regular expression “myocardial infarction” and the pharmacology description contains the regular expression “plasmin”. Union the results with the labels of entities where the general function descriptions contain the regular expression “cytokine” and the specific function descriptions contain the regular expression “antibacterial”.
3. Find the URIs and indication descriptions of entities where the indication descriptions contain the regular expression “'thrombocytopenia” and the biotransformation descriptions contain the regular expression “catabolic hydrolysis”
4. Find the distinct URIs, melting point descriptions and indication descriptions of entities where the generic names contain the regular expression “'Cetuximab”
5. Find the URIs of entities were the pharmacology descriptions contain the regular expression “Angiomax” and the half-life descriptions contain the regular expression “25 min”
6. Find the distinct homepages of entities where the mechanism of action descriptions contain the regular expression “metabolism” and the pharmacology descriptions contain the regular expression “diabetes”
7. Retrieve the RDF graphs of entities where the pharmacology descriptions contain the regular expression “cancer”
8. Retrieve the RDF graphs of entities where the synthesis reference descriptions contain the regular expression “Pfister”
9. Retrieve the RDF graphs of entities where the absorption descriptions contain the regular expression “azelastine”
10. Retrieve the RDF graphs of entities where the biotransformation descriptions contain the regular expression “'a-methyldopa mono-0-sulfate”
11. Retrieve the RDF graphs of entities where the generic names contain the regular expression

"Bromfenac"

12. Retrieve the RDF graphs of entities where the textual description of the entities contain the regular expression "anesthetic" and the indication/treatment descriptions contain the regular expression "analgesia"
13. Construct an RDF graph consisting of the mechanism of action descriptions of entities where the indication descriptions contain the regular expression "hemodynamic imbalances"
14. Construct an RDF graph consisting of the textual descriptions of entities where the indication/treatment descriptions contain the regular expression "atrial fibrillation" and the absorption descriptions contain the regular expression "reproducibly absorbed"
15. Construct an RDF graph consisting of the pharmacology descriptions of entities where the textual descriptions of the entities contain the regular expression "isomnia"
16. Construct an RDF graph consisting of the half-life descriptions of entities where the indication/treatment descriptions contain the regular expression "leukemia"
17. Construct an RDF graph consisting of the absorption descriptions where the half-life descriptions contain the regular expression "8 hours"
18. Construct an RDF graph consisting of the indication/treatment descriptions of entities where the labels contain the regular expression "Hydrocone"
19. Check if any of the entities have a mechanism of action description containing the regular expression "tumor cells"
20. Check if any of the entities have a textual description containing the regular expression "iodine"
21. Check if any of the entities have a mechanism of action description containing the regular expression "cardiac stimulation" and an indication/treatment description containing the regular expression "hypertension"
22. Check if any of the entities have state description containing the regular expression "Solid" and an indication/treatment description containing the regular expression "hypertension"
23. Check if any of the entities have a melting point description containing the regular expression "166-167"
24. Check if any of the entities have a pharmacology description containing the regular expression "phencyclidine" and a mechanism of action description containing the regular expression "NMDA receptor"

Table 4.4: The query mix for the DrugBank data set represented with SPARQL

SPARQL queries
1. SELECT DISTINCT ?label WHERE {?s drugbank:text ?text; rdfs:label ?label. Filter regex(?text, 'Interferon'). Filter regex(?text, 'theophylline')} 2. SELECT DISTINCT ?label WHERE {{?s drugbank:indication ?treatment; rdfs:label ?label; drugbank:pharmacology ?pharmacology . Filter regex(?treatment, 'myocardial infarction'). Filter regex(?pharmacology, 'plasmin')} UNION {?s drugbank:generalFunction ?generalFunction; drugbank:specificFunction ?specificFunction ; rdfs:label ?label. Filter regex(?generalFunction, 'cytokine'). Filter regex(?specificFunction , 'antibacterial')}} 3. SELECT ?s ?indication WHERE {?s drugbank:biotransformation ?biotransformation; drugbank:indication ?indication. Filter regex(?indication, 'thrombocytopenia'). Filter regex(?biotransformation, 'catabolic hydrolysis')} 4. SELECT DISTINCT ?s ?meltingPoint ?indication WHERE {?s drugbank:indication ?indication; drugbank:meltingPoint ?meltingPoint; drugbank:genericName ?genericName. Filter regex(?genericName, 'Cetuximab')} 5. SELECT ?s WHERE {?s drugbank:pharmacology ?pharmacology; drugbank:halfLife ?halfLife. Filter regex(?pharmacology, 'Angiomax'). Filter regex(?halfLife, '25 min')} 6. SELECT DISTINCT ?page WHERE {?s drugbank:mechanismOfAction ?mechanismOfAction; foaf:page ?page; drugbank:pharmacology ?pharmacology. Filter regex(?mechanismOfAction, 'metabolism'). Filter regex(?pharmacology , 'diabetes')} 7. DESCRIBE ?s WHERE {?s drugbank:pharmacology ?pharmacology. Filter regex(?pharmacology, 'cancer')} 8. DESCRIBE ?s WHERE {?s drugbank:synthesisReference ?synthesisReference. Filter regex(?synthesisReference, 'Pfister')} 9. DESCRIBE ?s WHERE {?s drugbank:absorption ?absorption. Filter regex(?absorption, 'azelastine')} 10. DESCRIBE ?s WHERE {?s drugbank:biotransformation ?biotransformation. Filter regex(?biotransformation, 'a-methyldopa mono-0-sulfate')} 11. DESCRIBE ?s WHERE {?s drugbank:genericName ?genericName. Filter regex(?genericName, 'Bromfenac')} 12. DESCRIBE ?s WHERE {?s drugbank:description ?description; drugbank:indication ?indication. Filter regex(?description, 'anesthetic'). Filter regex(?indication, 'analgesia')} 13. CONSTRUCT {?s drugbank:mechanismOfAction ?mechanismOfAction} WHERE {?s drugbank:mechanismOfAction ?mechanismOfAction; drugbank:indication ?indication. Filter regex(?indication, 'hemodynamic imbalances')}

14. CONSTRUCT {?s drugbank:description ?description} WHERE {?s drugbank:description ?description; drugbank:absorption ?absorption; drugbank:indication ?indication. Filter regex(?indication, 'atrial fibrillation'). Filter regex(?absorption, 'reproducibly absorbed')}
15. CONSTRUCT {?s drugbank:pharmacology ?pharmacology} WHERE {?s drugbank:pharmacology ?pharmacology; drugbank:description ?description. Filter regex(?description, 'insomnia')}
16. CONSTRUCT {?s drugbank:halfLife ?halfLife} WHERE {?s drugbank:halfLife ?halfLife; drugbank:indication ?indication. Filter regex(?indication, 'leukemia')}
17. CONSTRUCT {?s drugbank:absorption ?absorption} WHERE {?s drugbank:absorption ?absorption; drugbank:halfLife ?halfLife. Filter regex(?halfLife, '8 hours')}
18. CONSTRUCT {?s drugbank:indication ?indication} WHERE {?s drugbank:indication ?indication; rdfs:label ?label. Filter regex(?label, 'Hydrocodone')}
19. ASK {?s drugbank:mechanismOfAction ?mechanismOfAction. Filter regex(?mechanismOfAction, 'tumor cells')}
20. ASK {?s drugbank:description ?description. Filter regex(?description, 'iodine')}
21. ASK {?s drugbank:mechanismOfAction ?mechanismOfAction; drugbank:indication ?indication. Filter regex(?mechanismOfAction, 'cardiac stimulation'). Filter regex(?indication, 'hypertension')}
22. ASK {?s drugbank:state ?state; rdfs:label ?label. Filter regex(?state, 'Solid'). Filter regex(?label, 'Allylprodine')}
23. ASK {?s drugbank:meltingPoint ?meltingPoint. Filter regex(?meltingPoint, '166-167')}
24. ASK {?s drugbank:pharmacology ?pharmacology; drugbank:mechanismOfAction ?mechanismOfAction. Filter regex(?pharmacology, 'phencyclidine'). Filter regex(?mechanismOfAction, 'NMDA receptor')}

4.2.5.2 The query mix for the Geographical Coordinates (DBpedia) data set

This section will present the query mix for the Geographical Coordinates (DBpedia) data set. Table 4.5 shows the natural language queries, whereas Table 4.6 shows the natural language queries translated into a SPARQL representation.

Table 4.5: The query mix for the Geographical Coordinates (DBpedia) data set

Queries
1. Find the latitude and longitude of entities where the latitude is between 50 and 60 and the longitude is between 5 and 10
2. Find the URIs of geographical locations where the latitude equals or is less than 50 and equals or is higher than 49, and the longitude equals or is higher than 9 and equals or is less than 10
3. Find the URIs of geographical locations where the latitude minus 37.785834 is smaller than, or equals 0.04 and 37.785834 minus the latitude is smaller than, or equals 0.04, and the longitude minus -122.406417 is smaller than, or equals 0.04 and -122.406417 minus the longitude is smaller than, or equals 0.04
4. Find the URIs of geographical locations where the latitude times 2 is higher than 96 and the latitude divided by 3 is higher than 15, and the longitude divided by 2 is higher than 4 and the longitude times 2 is less than 30.
5. Find the URIs of geographical locations where the latitude is higher than 60 or the latitude is higher than 50, and the longitude is higher than 1
6. Find the URIs of geographical locations where the latitude minus 50 is less than or equals 0.04, and 50 minus the latitude is less than or equals 0.04, and the longitude minus 0 is less than or equals 0.04, and 0 minus the longitude is less than or equals 0.04
7. Retrieve the RDF graphs of geographical locations where the latitude minus 10 is less than or equals 50, and the latitude is higher than 40, and the longitude is between 5 and 10
8. Retrieve the RDF graphs of geographical locations where the latitude times 3 is higher than 150 and the latitude divided by 2 is higher than 25, and the longitude minus 2 is higher than 5
9. Retrieve the RDF graphs of geographical locations where the latitude is between 50 and 80, and the longitude is between 0 and 10
10. Retrieve the RDF graphs of geographical locations where the latitude equals or is less than 30, and the latitude equals or is higher than 20
11. Retrieve the RDF graphs of geographical locations where the longitude is between 10 and 12
12. Retrieve the RDF graphs of geographical locations where the latitude divided by 3 equals or is less than 20, the latitude times 2 is less than 100, the longitude times 2 is higher than 20, and the longitude times 2 is less than 30
13. Construct an RDF graphs of geographical locations, consisting of the triple ?s grs:point (?lat ?long), where the latitude minus 20 equals or is less than 40, the latitude is higher than 30, and the longitude is between 7 and 15
14. Construct an RDF graphs of geographical locations, consisting of the triple ?s grs:point (?lat

?long), where the latitude times 2 is higher than 100, and the longitude divided by 3 is higher than 3

15. Construct an RDF graphs of geographical locations, consisting of the triple ?s grs:point (?lat ?long), where the latitude divided by 5 equals or is smaller than 10, and the longitude plus 20 is higher than 30
16. Construct an RDF graph of every geographical location, consisting of the triple ?s geo:geometry (?long ?lat), where the latitude is between 10 and 20, and the longitude is between 0 and 10
17. Construct an RDF graph of every geographical location, consisting of the triple ?s geo:geometry (?long ?lat), where the latitude is between 30 and 50, and the longitude is between 18 and 20
18. Construct an RDF graph of every geographical location, consisting of the triple ?s geo:geometry (?long ?lat), where the latitude minus 20 is higher than 30, the latitude plus 2 is higher than 50, and the longitude is between 10 and 15
19. Check if any of the geographical locations have a latitude between 40 and 41
20. Check if any of the geographical locations have a longitude that equals or is less than 10, and a latitude that equals or is higher than 50
21. Check if there exist any geographical locations where 50 minus the latitude is higher than 10, and the longitude times 2 is higher than 20
22. Check if there exist any geographical locations where the longitude minus 10 equals or is less than 0.5
23. Check if any of the geographical locations have a latitude that equals or is higher than 60, and a longitude that equals or is less than 20
24. Check if there exist any geographical locations where the longitude divided by 2 equals or is less than 5, and the latitude plus 20 is higher than 70

Table 4.6: The query mix for the Geographical Coordinates (DBpedia) data set represented with SPARQL

SPARQL queries
1. SELECT ?lat ?long WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat > 50 && ?lat < 60) && (?long > 5 && ?long < 10))} 2. SELECT ?s WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat <= 50 && ?lat >= 49) && (?long >= 9 && ?long <= 10))} 3. SELECT ?s WHERE {?s geo:lat ?lat; geo:long ?long. Filter ((xsd:double(?lat) - 37.785834

<= 0.040000) && (37.785834 - xsd:double(?lat) <= 0.040000) && (xsd:double(?long) - -122.406417 <= 0.040000) && (-122.406417 - xsd:double(?long) <= 0.040000))}

4. SELECT ?s WHERE {?s geo:lat ?lat; geo:long ?long . Filter((xsd:double(?lat) * 2 > 96) && (xsd:double(?long) / 2 > 4) && (xsd:double(?lat) / 3 > 15) && (xsd:double(?long) * 2 < 30))}
5. SELECT ?s WHERE { ?s geo:lat ?lat; geo:long ?long. Filter((?lat > xsd:float("60") || ?lat > xsd:float("50")) && ?long > xsd:float("1"))}
6. SELECT ?s WHERE {?s geo:lat ?lat; geo:long ?long . Filter((xsd:float(?lat) - 50 <= 0.40000) && (50 - xsd:float(?lat) <= 0.40000) &&(xsd:float(?long) - 0 <= 0.40000) && (0 - xsd:float(?long) <= 0.40000))}
7. DESCRIBE ?s WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat - 10 <= 50 && ?lat > 40) && (?long > 5 && ?long < 10))}
8. DESCRIBE ?s WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat * 3 > 150 && ?lat / 2 > 25) && (?long - 2 > 5))}
9. DESCRIBE ?s WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat > 50 && ?lat < 80) && (?long < 10 && ?long > 0))}
10. DESCRIBE ?s WHERE {?s geo:lat ?lat. Filter(?lat <= 30 && ?lat >= 20)}
11. DESCRIBE ?s WHERE {?s geo:long ?long. Filter(?long > 10 && ?long < 12)}
12. DESCRIBE ?s WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat / 3 <= 20 && ?lat * 2 < 100) && (?long * 2 > 20 && ?long * 2 < 30))}
13. CONSTRUCT {?s grs:point (?lat ?long) } WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat - 20 <= 40 && ?lat > 30) && (?long > 7 && ?long < 15))}
14. CONSTRUCT {?s grs:point (?lat ?long) } WHERE {?s geo:lat ?lat; geo:long ?long. Filter(?lat * 2 > 100 && ?long / 3 > 3)}
15. CONSTRUCT {?s grs:point (?lat ?long) } WHERE {?s geo:lat ?lat; geo:long ?long. Filter(?lat / 5 <= 10 && ?long + 20 > 30)}
16. CONSTRUCT {?s geo:geometry (?long ?lat) } WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat < 20 && ?lat > 10) && (?long < 10 && ?long > 0))}
17. CONSTRUCT {?s geo:geometry (?long ?lat) } WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat < 50 && ?lat > 30) && (?long < 20 && ?long > 18))}
18. CONSTRUCT {?s geo:geometry (?long ?lat) } WHERE {?s geo:lat ?lat; geo:long ?long. Filter((?lat - 20 > 30 && ?lat + 2 > 50) && (?long > 10 && ?long < 15))}
19. ASK {?s geo:lat ?lat. Filter(?lat > 40 && ?lat < 41)}
20. ASK {?s geo:lat ?lat; geo:long ?long. Filter(?long <= 10 && ?lat >= 50)}
21. ASK {?s geo:lat ?lat; geo:long ?long. Filter(50 - ?lat > 10 && ?long * 2 > 20)}
22. ASK {?s geo:long ?long. Filter(?long - 10 <= 0.5)}

23. ASK {?s geo:lat ?lat; geo:long ?long. Filter(?lat >= 60 && ?long <= 20)}
24. ASK {?s geo:lat ?lat; geo:long ?long. Filter(?long / 2 <= 5 && ?lat + 20 > 70)}

4.2.6 Rules

This section will define the rules for the entire benchmark evaluation cycle. The benchmark evaluation shall include

- three iterations for each divided data set in both use-cases (approximately 100 000 triples, 350 000 triples, 700 000 triples), where the test-machine and both SUT will be shut down and restarted between each iteration (the query mix for each use-case will thus be executed nine times in total for each use-case; three times for each divided data set. This will be the basis for an average query-execution time on each query and query mix)
- the execution of warm-up queries to warm up the stores for 30 minutes, in order to simulate normal working conditions for the data stores
- a logging mechanism in the test-driver that keeps track of all relevant statistics

Chapter 5: Results

This chapter will present the results of the benchmark evaluation of the two benchmark test use-cases described in section 4.2: the DrugBank regular expression filtering use-case and the Geographical Coordinates numerical/logical filtering use-case. The results are analyzed based on the metrics and definitions presented in section 4.2. All results presented in this chapter are based on the aQET of each query in the two triplestores, meaning that the query execution time of each query presented in the tables are an average of three query executions of each query.

As mentioned in section 4.2.3, both the DrugBank data set and the Geographical Coordinates data set were divided into three different versions for the benchmark evaluation. All the different versions of the data set were first loaded into the Joseki triplestore, and a warm-up query mix consisting of ten queries with different query forms, with and without filter clauses, was executed ten times over the given data sets. After this, the query mixes presented in section 4.2.5.1 and section 4.2.5.2 were executed over the data sets three times. Between each execution of the query mix, the SUT was restarted and the warm-up query mix was once again executed ten times before continuing with the next iteration of the evaluation query mix. All results were recorded, and the same procedure was repeated with the FILT triplestore.

This section will refer to each of the data set sizes described in section 4.2.3 as “S” for the smallest data set version, “M” for the medium data set version, and “L” for the large data set, consisting of the entire data set. The results from the DrugBank data set and the Geographical Coordinates data set were each analyzed in a separate, two way analysis of variance (ANOVA) with the factors Size (S, M, L) and Store (FILT, Joseki). The critical values for F will be reported in the results with the signifiers described in Table 5.1.

Table 5.1: The probability numbers and their signifiers

Probability number (p)	Probability signifier
0.001 (0.1%)	***
0.01 (1%)	**
0.05 (5%)	*

The overall results of the DrugBank regular expression filtering use-case can be seen in Figure 5.1. The chart shows that FILT outperformed Joseki when executing SELECT queries, but came short when executing queries of the other query forms.

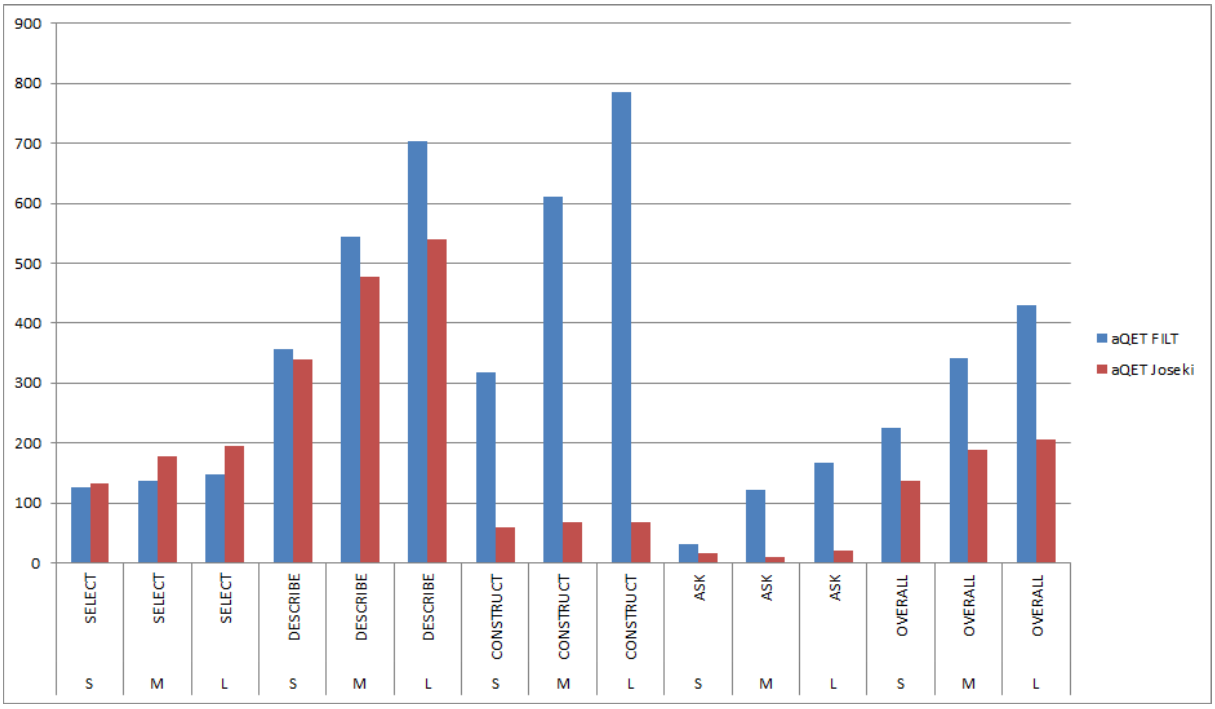


Figure 5.1: The overall benchmark results of the DrugBank regular expression filtering use-case

Figure 5.2 shows the overall results of the Geographical Coordinates numerical/logical filtering use-case. The chart shows that FILT outperformed Joseki to a great extent for all query forms except ASK queries.

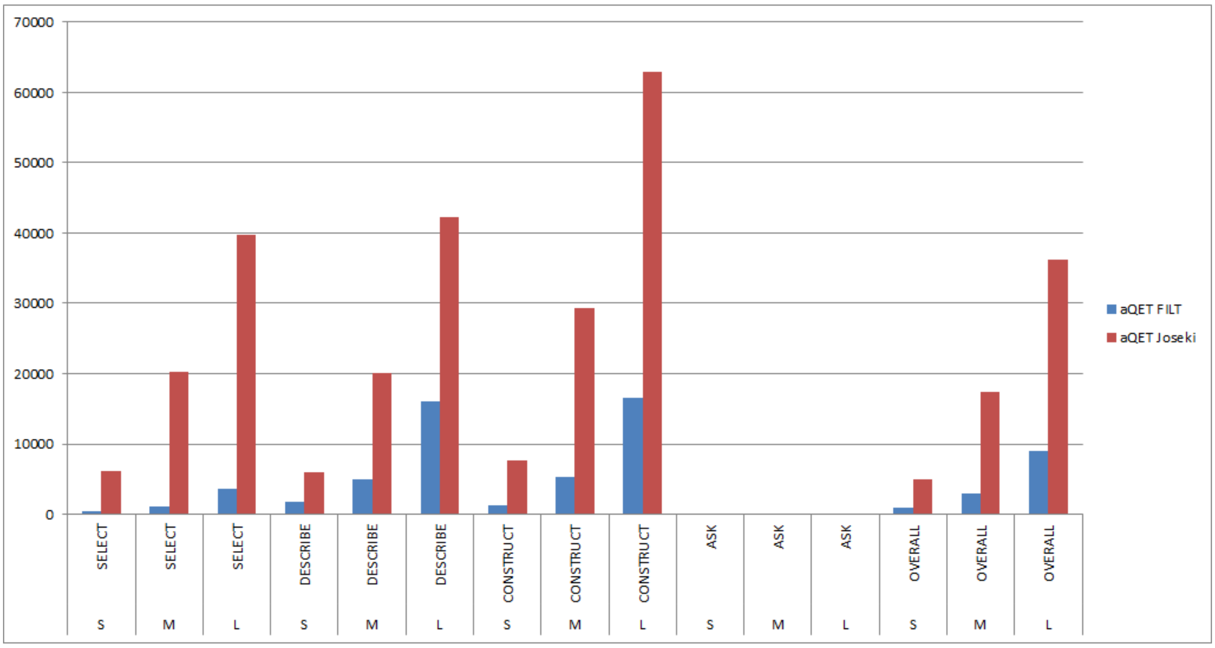


Figure 5.2: The overall benchmark results of the Geographical Coordinates numerical/logical filtering use-case

The following pages will present the individual comparisons and statistical results of the DrugBank regular expression filtering use-case and the Geographical Coordinates numerical/logical filtering use-case.

5.1 SPARQL regex filtering in the DrugBank data set

The results of the DrugBank use-case indicate that the SELECT queries of the query mix had a significant difference in the results of FILT and Joseki. Figure 5.2 shows that FILT performs faster than Joseki with SELECT regex queries for all data set sizes. The results indicate that the larger the data set is, Joseki performs significantly worse, as opposed to FILT that more or less performs in the same way regardless of data set size, with small differences in the aQET.

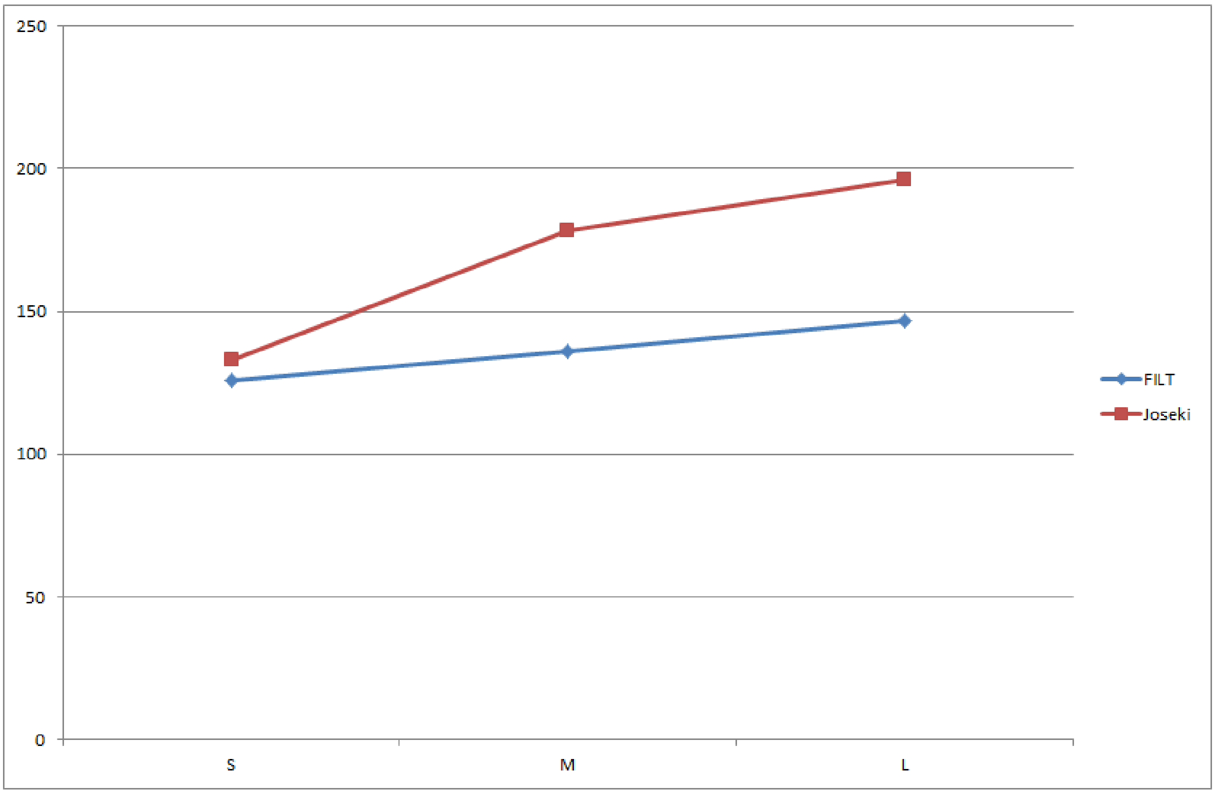


Figure 5.2: The aQET of the SPARQL SELECT queries in the query mix in both FILT and Joseki

Looking at the probability numbers in Table 5.2, it is evident that the data set size (Size) is a significant factor when executing the SELECT queries in both triplestores, with $p < 0.01$. The difference between the two triplestores (Store) is also a significant factor, with $p < 0.001$. The interaction between the data set sizes and the triplestores (Size:Store) is not significant, with $p < 0.10$.

Table 5.2: The statistics summary of the execution of the SELECT queries in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	10.969	0.001954 **
Store	1	19.382	0.000862 ***
Size:Store	2	3.107	0.081787

Figure 5.3 shows that, as opposed to the results of the SELECT queries, FILT and Joseki performed almost similar on the small data set size (S) when executing the DESCRIBE queries, with Joseki having a slight advantage. However, as the data set size increased Joseki performed faster than FILT.

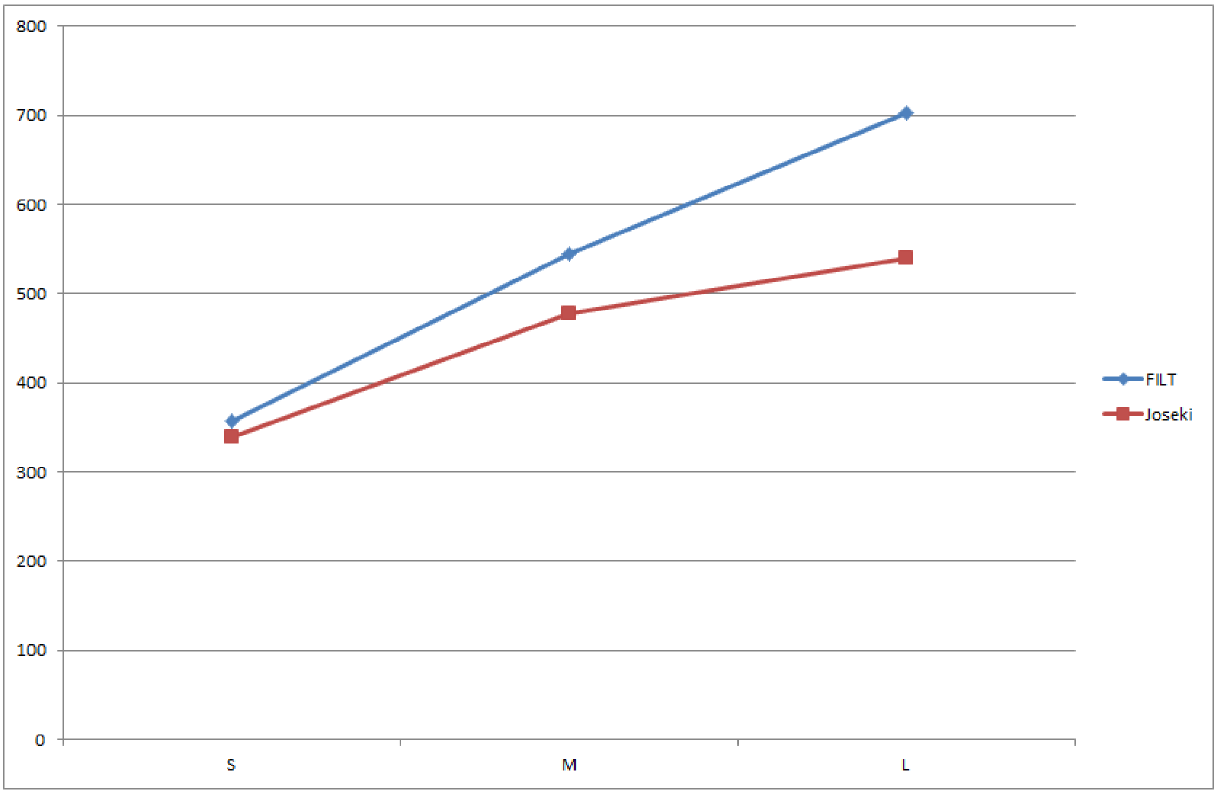


Figure 5.3: The aQET of the DESCRIBE queries in the query mix in both FILT and Joseki

The probability numbers in Table 5.3 shows that the data set size (Size) is a significant factor when executing the DESCRIBE queries in both triplestores, with p < 0.001. The difference between the two triplestores (Store) is also a significant factor, with p < 0.001. The interaction between the data set sizes and the triplestores (Size:Store) is also significant, with p < 0.01.

Table 5.3: The statistics summary of the execution of the DESCRIBE queries in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	159.60	2.26e-09 ***
Store	1	43.07	2.68e-05 ***
Size:Store	2	11.88	0.00143 **

Figure 5.4 shows how both FILT and Joseki performed when executing the CONSTRUCT queries over the different data set sizes. These results show that Joseki performed better than FILT when executing the CONSTRUCT queries, regardless of the data set size. As the data set size increased FILT performed worse, whereas Joseki performed more or less the same for all data set sizes.

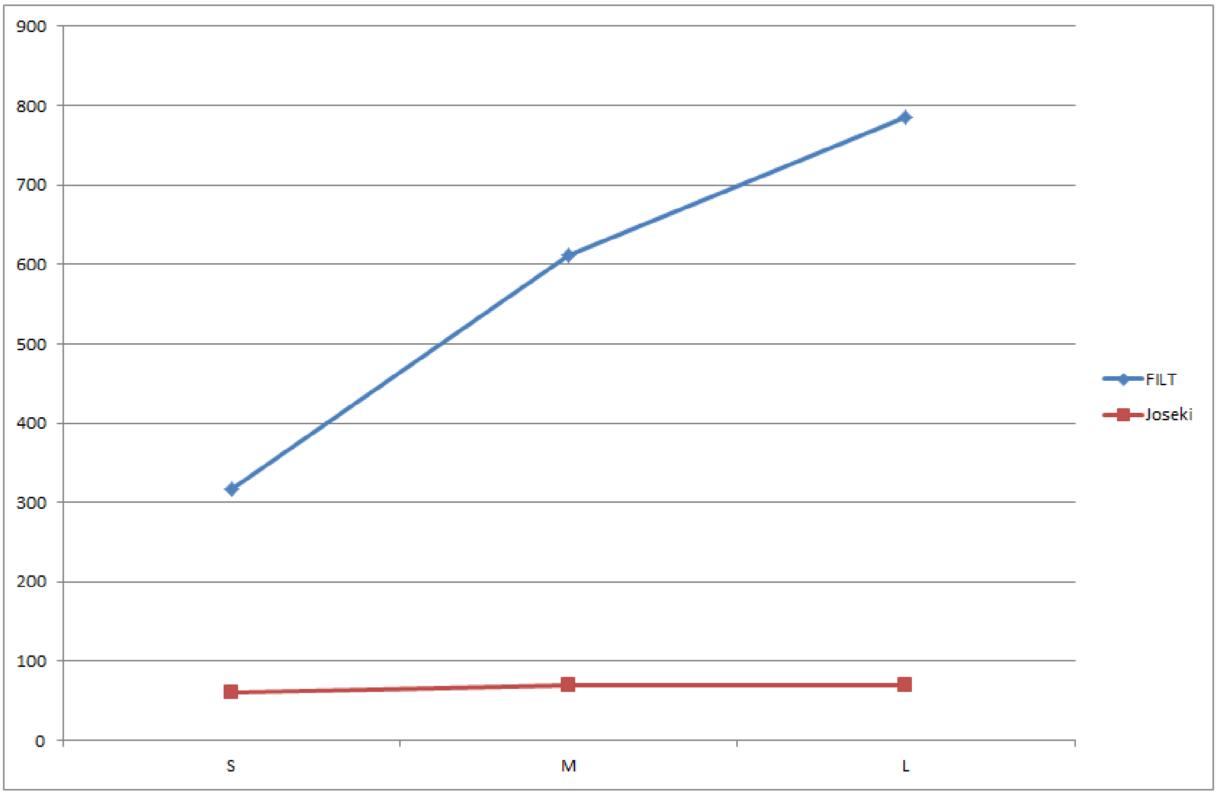


Figure 5.4: The aQET of the CONSTRUCT queries in the query mix in both FILT and Joseki

Looking at the probability numbers in Table 5.4, it is evident that the data set size (Size) is a significant factor when executing the CONSTRUCT queries in both triplestores, with p < 0.001. The difference between the two triplestores (Store) is also a significant factor, with p < 0.001. The interaction between the data set sizes and the triplestores (Size:Store) is also significant, with a p < 0.001.

Table 5.4: The statistics summary of the execution of the CONSTRUCT queries in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	902.6	8.29e-14 ***
Store	1	11814.8	< 2e-16 ***
Size:Store	2	828.5	1.38e-13 ***

Figure 5.5 shows that Joseki clearly performed better than FILT when executing the ASK queries. FILT executed the ASK queries slower as the data set size increased, whereas there were minimal differences in the aQET of Joseki as the data set size increased. Despite Joseki executing the ASK queries faster than FILT, the largest difference between the aQET of Joseki and FILT when executing the ASK queries were 145 milliseconds.

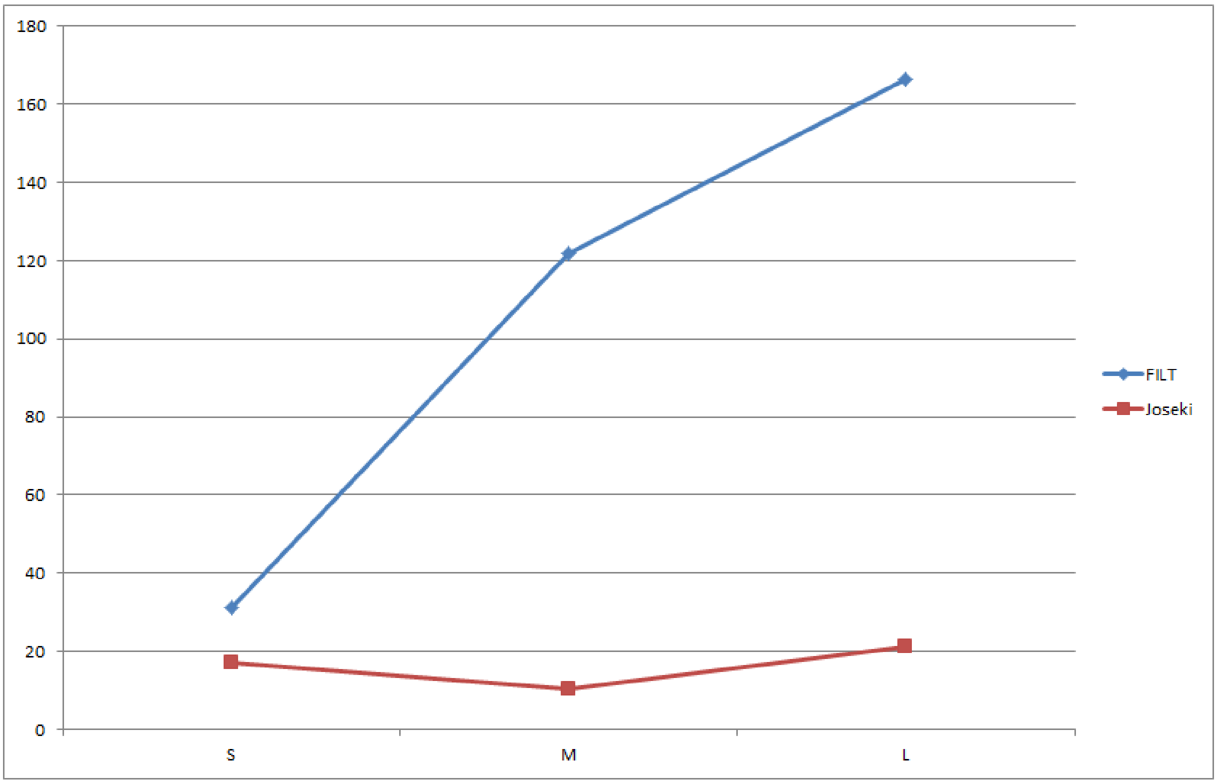


Figure 5.5: The aQET of the ASK queries in the query mix in both FILT and Joseki

Looking at the probability numbers in Table 5.5, it is evident that the data set size (Size) is not a significant factor when executing the ASK queries in both triplestores, with p = 0.662. The difference between the two triplestores (Store) is highly significant, with p < 0.001. The interaction between the data set sizes and the triplestores (Size:Store) is not significant, with p = 0.076.

Table 5.5: The statistics summary of the execution of the ASK queries in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	0.427	0.662
Store	1	170.144	1.9e-08 ***
Size:Store	2	3.220	0.076

Based on the results of the different query forms in the query mix, Figure 5.6 shows the overall aQET of all queries in the query mix. It is clear that Joseki performs faster than FILT to a great extent, and the difference is bigger as the data set size increases. FILT performed faster than Joseki for the SELECT queries, but for the other three query forms Joseki performed faster than FILT.

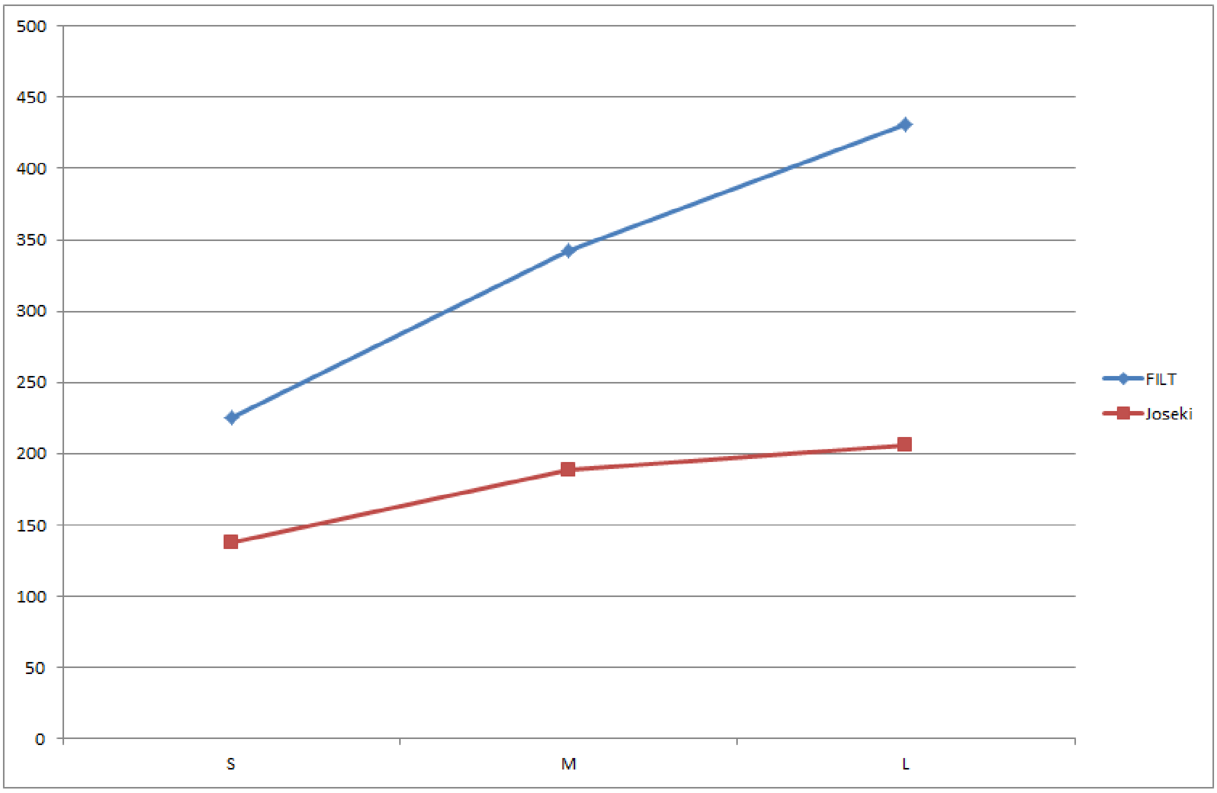


Figure 5.6: The aQET of the all queries in the query mix in both FILT and Joseki

Looking at the probability numbers in Table 5.6, it is evident that the data set size (Size) is a significant factor when executing the entire query mix in both triplestores, with p < 0.001. The difference between the two triplestores (Store) is also a significant factor, with p < 0.001. The interaction between the data set sizes and the triplestores (Size:Store) is also significant, with $p < 0.001$.

Table 5.6: The statistics summary of the execution of the entire query mix in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	472.6	3.88e-12 ***
Store	1	1771.6	2.10e-14 ***
Size:Store	2	118.1	1.28e-08 ***

To summarize the SPARQL regex filter query use-case, FILT outperforms Joseki when it comes to SELECT queries. The results also show that Joseki performs faster than FILT with the other query forms: DESCRIBE, CONSTRUCT and ASK.

5.2 SPARQL numerical/ logical filtering in the Geographical Coordinates data set (DBpedia)

The results of the Geographical Coordinates use-case clearly show that the SELECT queries of the query mix had a significant difference in the results of FILT and Joseki. Figure 5.7 shows that FILT performed remarkably faster than Joseki for the six SELECT queries in the query mix. The difference between FILT and Joseki for the small data set (S), consisting of 250,000 triples, were noteworthy, and as the data set size increased FILT performs significantly faster than Joseki. The biggest difference in the aQET of the SELECT queries occurred when executing the queries over the large data set (L), consisting of 1,700,000 triples, where FILT executed the SELECT queries more than 35,000 milliseconds (35 seconds) faster than Joseki.

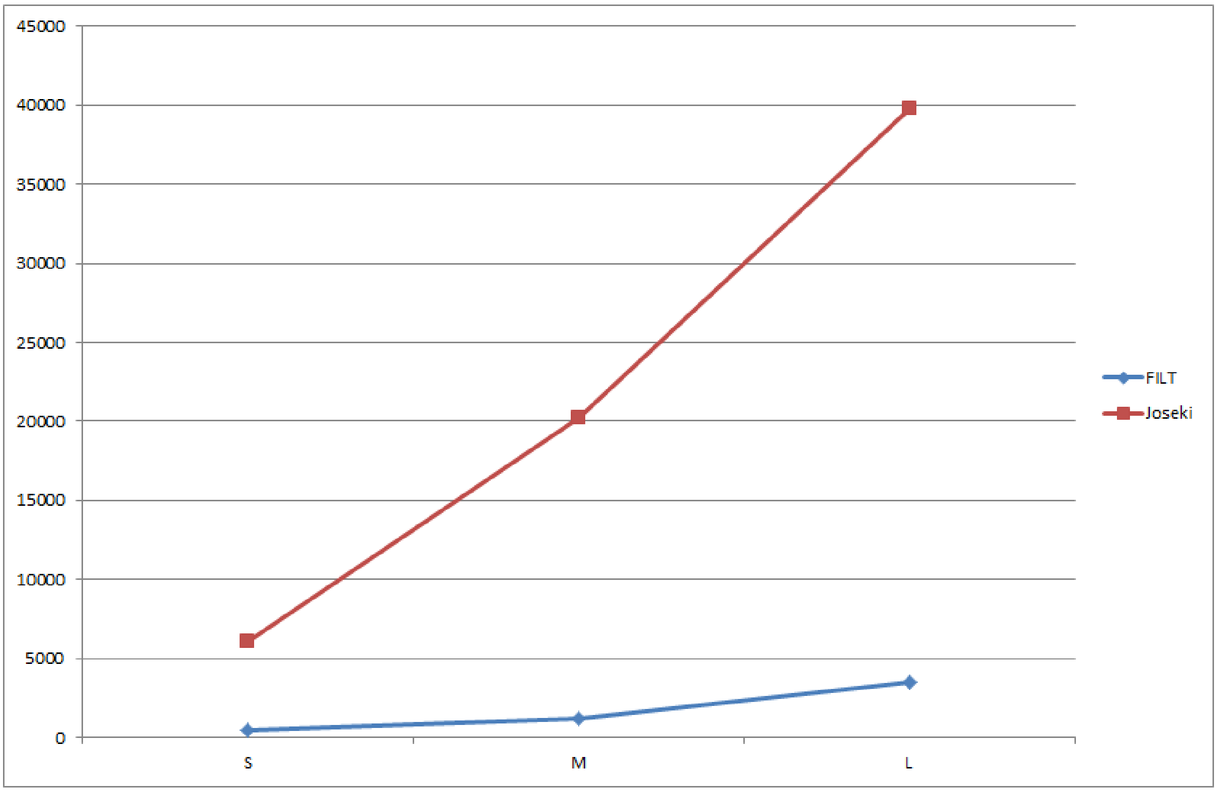


Figure 5.7: The aQET of the SELECT queries in the query mix in both FILT and Joseki

Looking at the probability numbers in Table 5.7, it is evident that the data set size (Size) is a significant factor when executing the SELECT queries in both triplestores, $p < 0.001$. The difference between the two triplestores (Store) is also a significant factor, $p < 0.001$. The interaction between the data set sizes and the triplestores (Size:Store) is also significant, $p < 0.001$.

Table 5.7: The statistics summary of the execution of the SELECT queries in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	62126	<2e-16 ***
Store	1	224788	<2e-16 ***
Size:Store	2	42901	<2e-16 ***

Figure 5.8 shows the performance of FILT and Joseki when executing the DESCRIBE queries. The chart indicates that there is a similarity between the aQET of SELECT queries and DESCRIBE queries in both FILT and Joseki. However, both FILT and Joseki performed faster when executing the SELECT queries compared to DESCRIBE queries. The difference of the aQET between FILT and Joseki were significant when executing the DESCRIBE queries. The biggest difference in the aQET of

the DESCRIBE queries occurred when executing the DESCRIBE queries over the large data set (L), consisting of 1,700,000 triples, with a time difference of 27,000 milliseconds (27 seconds).

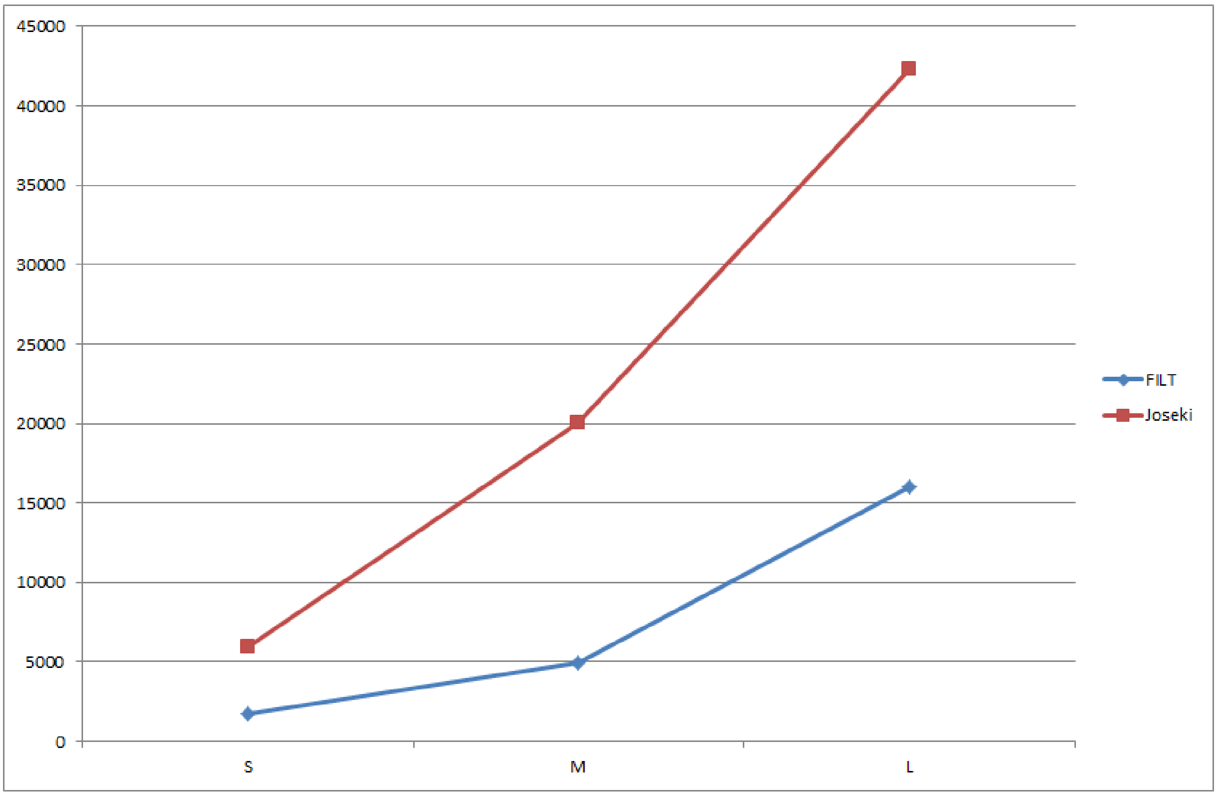


Figure 5.8: The aQET of the DESCRIBE queries in the query mix in both FILT and Joseki

Looking at the probability numbers in Table 5.8, it is evident that the data set size (Size) is a significant factor when executing the DESCRIBE queries in both triplestores, p < 0.001. The difference between the two triplestores (Store) is also a significant factor, p < 0.001. The interaction between the data set sizes and the triplestores (Size:Store) is also significant, p < 0.001.

Table 5.8: The statistics summary of the execution of the DESCRIBE queries in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	144370	<2e-16 ***
Store	1	151663	<2e-16 ***
Size:Store	2	26506	<2e-16 ***

Figure 5.9 shows how both FILT and Joseki performed when executing the CONSTRUCT queries over the different data set sizes. The results clearly indicate that FILT performed better than Joseki when executing the CONSTRUCT queries, regardless of the data set size. The biggest difference in the aQET of the two CONSTRUCT queries occurred when executing the CONSTRUCT queries over

the large data set (L), consisting of 1,700,000 triples, with a time difference of 46,000 milliseconds (46 seconds).

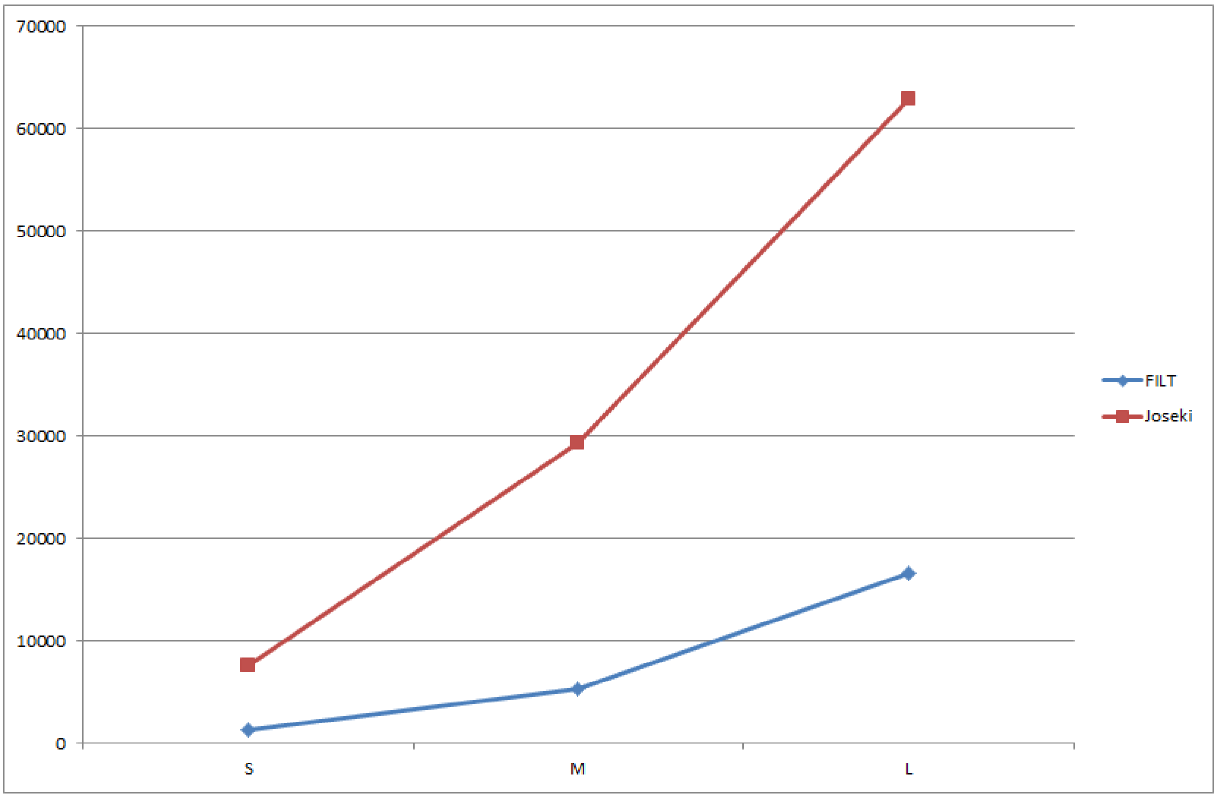


Figure 5.9: The aQET of the CONSTRUCT queries in the query mix in both FILT and Joseki

Looking at the probability numbers in Table 5.9, it is evident that the data set size (Size) is a significant factor when executing the CONSTRUCT queries in both triplestores, $p < 0.001$. The difference between the two triplestores (Store) is also a significant factor, $p < 0.001$. The interaction between the data set sizes and the triplestores (Size:Store) is also significant, $p < 0.001$.

Table 5.9: The statistics summary of the execution of the CONSTRUCT queries in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	27411	<2e-16 ***
Store	1	42151	<2e-16 ***
Size:Store	2	8666	<2e-16 ***

Figure 5.10 shows Joseki executed the ASK queries faster than FILT, regardless of data set size. However, there is an indication that FILT performs faster as the data set size increases, whereas Joseki performs slower as the data set size increases. Moreover, despite FILT performing slower when

executing the ASK queries, the results indicate that FILT eventually would perform faster than Joseki as the data set size increased even further.

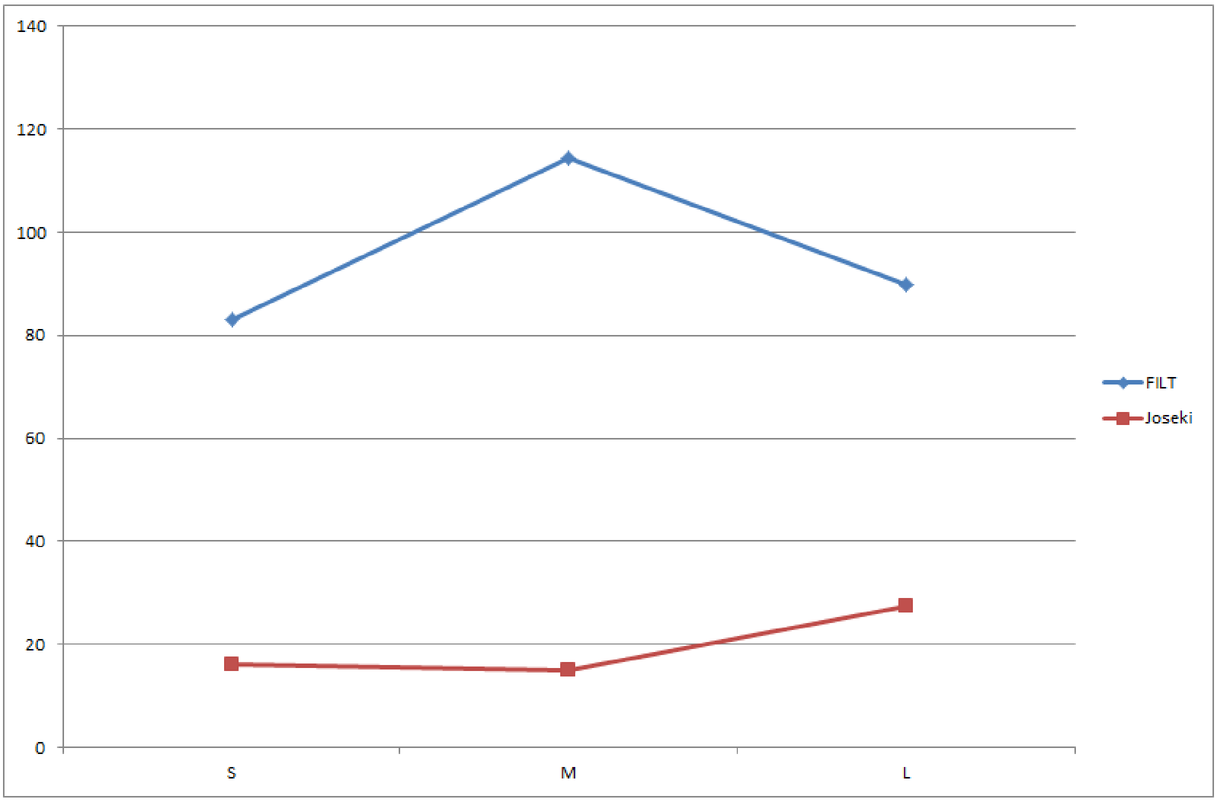


Figure 5.10: The aQET of the ASK queries in the query mix in both FILT and Joseki

Looking at the probability numbers in Table 5.10, it is evident that the data set size (Size) is a significant factor when executing the ASK queries in both triplestores, with p < 0.001. The difference between the two triplestores (Store) is also a significant factor, with p < 0.001, and finally the interaction between the data set sizes and the triplestores (Size:Store) is also significant, with p < 0.001.

Table 5.10: The statistics summary of the execution of the ASK queries in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	93.70	4.75e-08 ***
Store	1	1457.83	6.70e-14 ***
Size:Store	2	91.51	5.43e-08 ***

Based on the results of the different query forms in the query mix, Figure 5.11 shows the overall aQET of all queries in the query mix.

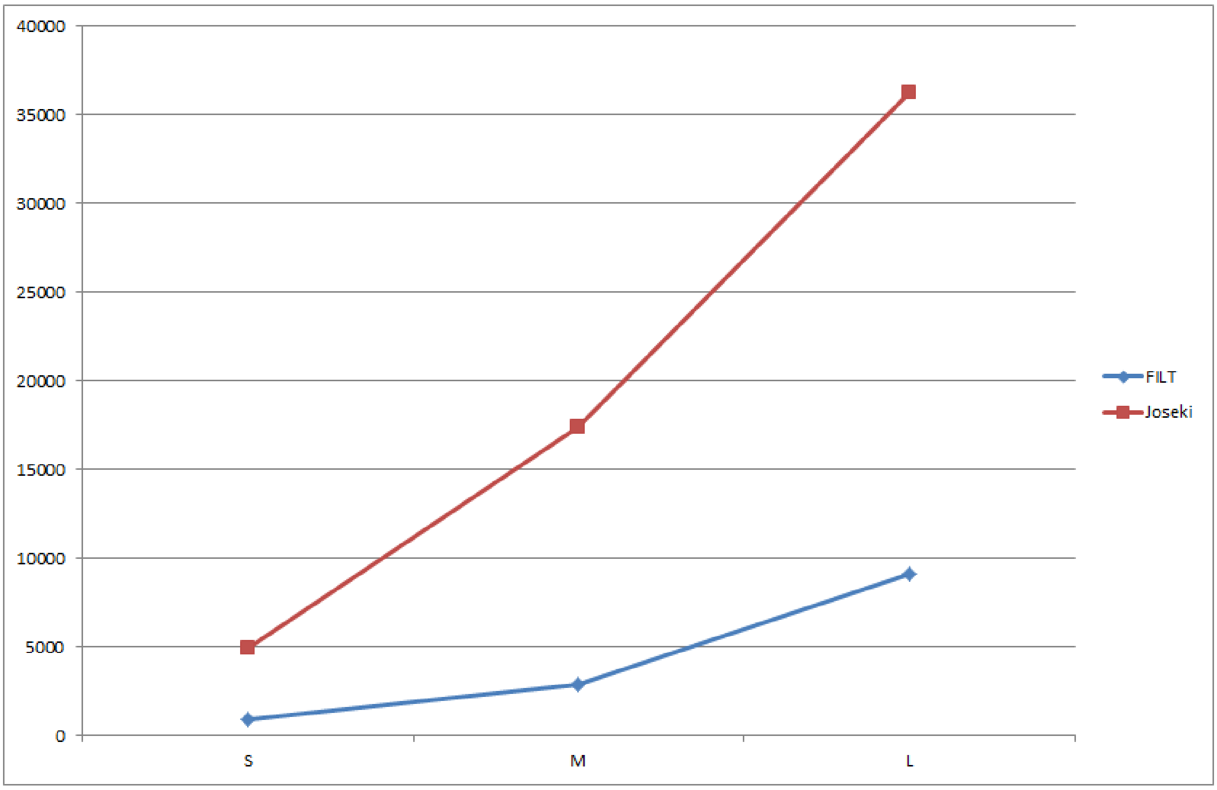


Figure 5.11: The aQET of the all queries in the query mix in both FILT and Joseki

Looking at the probability numbers in Table 5.11, it is evident that the data set size (Size) is a significant factor when executing the entire query mix in both triplestores, p < 0.001. The difference between the two triplestores (Store) is also a significant factor, p < 0.001. The interaction between the data set sizes and the triplestores (Size:Store) is also significant, p < 0.001.

Table 5.11: The statistics summary of the execution of the entire query mix in FILT and Joseki

	Df	F value	Pr(>F)
Size	2	159034	<2e-16 ***
Store	1	277308	<2e-16 ***
Size:Store	2	53541	<2e-16 ***

To summarize the SPARQL numerical/logical filter query use-case, FILT outperforms Joseki to a great extent with all query forms, except ASK queries. The biggest difference in the aQET between FILT and Joseki occurred when executing the query mix over the large data set (L), where FILT performed 28 milliseconds (28 seconds) faster than Joseki. The biggest difference for any of the query forms occurred when executing the CONSTRUCT queries, where FILT executed the queries 46 seconds faster than Joseki for the large data set.

Chapter 6: Discussion

The results of the benchmark evaluation show that FILT outperforms Joseki on SELECT queries in both use cases. In addition, every query form apart from the ASK queries was performed significantly faster with FILT than by Joseki in the SPARQL numerical/logical filter query use-case. However, this was not the case with the with the SPARQL regular expression filter query use-case, as Joseki performed faster than FILT with the DESCRIBE, CONSTRUCT and ASK query forms.

The results of the ASK, CONSTRUCT and DESCRIBE queries in the query mix of the SPARQL regular expression filter use-case affected the overall results of the use-case to a great extent, despite the aQET of the SELECT queries being faster in FILT than Joseki. The overall results clearly show that Joseki outperforms FILT when executing SPARQL regex filter queries of all query forms. There are no obvious explanations as to why FILT is faster than Joseki when executing SPARQL SELECT regex filter queries, yet slower when executing queries of the three other query forms. It is worth mentioning that even though Joseki performs better than FILT for the CONSTRUCT, DESCRIBE and ASK query forms, the differences in the aQET between Joseki and FILT are so small that they are hardly noticeable in a real-world querying scenario unless the times are actually recorded. This means that it is hard to locate any noticeable factors in the architecture of FILT that can lead to the aQET of the three query forms being slower than Joseki. However, there are some aspects in the way FILT returns query results that are worth discussing in light of the different outcomes of the four SPARQL query forms.

FILT executes all query forms in the exact same manner; the SPARQL filter clauses are being executed through Lucene, and the general SPARQL query is being executed through the Jena SPARQL processing engine. However, the difference in the way FILT returns query results from SELECT queries on one hand, and DESCRIBE and CONSTRUCT queries on the other hand, is that the results of the DESCRIBE and CONSTRUCT queries are converted from a Jena RDF model to a text string containing the raw RDF data, whereas SELECT queries are merely returned a SPARQL XML result set. Converting the Jena RDF model to a text string containing the raw RDF data is necessary in order to send the result object across the HTTP protocol, as a raw Jena RDF model cannot be sent through the HTTP protocol. This process is not time-consuming, but in many cases the time being spent by this conversion procedure is enough for FILT to return the results of the DESCRIBE and CONSTRUCT queries slower than Joseki, meaning that the aQET will be slower. It is likely that this conversion process is a major cause to the disadvantage FILT has compared to Joseki when executing DESCRIBE- and CONSTRUCT regex queries. For the SPARQL numerical/logical filter query case, the conversion process would not have a significant outcome on the results, because

Joseki was already executing the queries several seconds slower than FILT. Moreover, a couple of hundred milliseconds spent on converting the results are not noticeable in the SPARQL numerical/logical filter query use-case. Optimizing the process of returning results from DESCRIBE and CONSTRUCT queries in FILT are worth having a closer look at if FILT should be developed further.

ASK queries are constructed to check if the graph patterns and functions in the queries exists or do not exists in the data set. FILT copes with ASK queries the same way it copes with all the other query forms; the filter clauses are executed through Lucene and the general SPARQL query is executed through the Jena SPARQL processing engine. FILT does not retrieve all the entities that match the filter clauses executed through Lucene, but merely one of the entities. This is because as long as one entity corresponds to the filter clauses in the ASK query, this is enough for the filter clauses to be true. The entity is then being loaded into a local RDF model where the general SPARQL query is being executed. The results are finally returned as a SPARQL XML result set with a true or false binding. In FILT this is the most obvious and efficient way to deal with ASK queries discovered in this project, and it is difficult to say why Joseki outperforms FILT when it comes to all ASK queries, regardless of the two different use-cases. Finally, it is still worth mentioning that the highest time difference between FILT and Joseki with all ASK queries is only 145 milliseconds, which is hardly noticeable in a real-world querying scenario. Also, the results of the ASK queries executed in the SPARQL numerical/logical filter use-case indicate that FILT will eventually execute the ASK queries faster if the data set size increases further (see Table 5.9).

A final aspect worth discussing is the index structure of FILT and the variety of Lucene queries that are executed depending on what the SPARQL filter clauses of a query represent. The index structure in terms of document field analyzers and the entire indexer itself (Lucene provides several different indexing classes) may be factors that to some extent can provide answers as to why there are significant differences between the two use-cases. Also, the SPARQL regular expression filter clauses are executed through the Lucene RegexQuery class, whereas SPARQL numerical/logical filter clauses are mainly executed through the NumericRangeQuery, meaning that it is possible that the two Lucene query types have entirely different ways of filtering through data, and that one of them may be considerably faster than the other.

To summarize, in the SPARQL numerical/logical filter query use-case the overall results show that FILT outperforms Joseki to a great extent. However, for the SPARQL regex filter query use-case FILT only outperforms Joseki with SELECT queries, but are slower than Joseki for all other query forms. This means that Joseki has a faster overall performance than FILT for all the query forms combined. The fact that Joseki struggles to a great extent with SPARQL numerical/logical filter queries compared to SPARQL regex filter queries suggests that the major strength of Joseki lies in coping with SPARQL regex filter queries. FILT however, copes much better with SPARQL regular expression queries than Joseki does with SPARQL numerical/logical filter queries. This means that the weakness of FILT is much less significant and noticeable than the weakness of Joseki. Also, if the results of both use-cases were combined into one huge result set, FILT would outperform Joseki to a great extent, based on the fact that even though FILT performs slightly slower than Joseki in the SPARQL regex filter query use-case the query execution times are still very low (in most cases the aQET does not even reach a whole second). Finally, a conclusion can be drawn stating that FILT is a solution that should be used for executing SPARQL SELECT regex filter queries and SPARQL numerical/logical filter queries of all query forms.

Chapter 7: Conclusions and future work

The first research question of this project was to find out if the query-execution time of SPARQL filter queries could be decreased by storing RDF data and executing SPARQL filter queries through the Apache Lucene Framework. In order to measure this, two success criteria were designed. The first criterion stated that all general SPARQL queries without filter clauses, as well as SPARQL filter regex- and logical/numerical expression filter queries containing simple graph patterns and filter clauses should be executable through the system. This criterion was accomplished by implementing FILT, a system that sends general SPARQL queries without filter clauses directly to a conventional triplestore and retrieving the results, whereas SPARQL queries containing filter clauses are executed through a hybrid architecture consisting of the Apache Lucene and the Apache Jena frameworks. The second success criterion stated that SPARQL regex- and logical/numerical expression filter queries should execute faster through the system than through a conventional triplestore. FILT confirmed the first research question and its two related success criteria in spectacular fashion, outperforming Joseki with a time-difference up to 46 seconds.

The second research question aimed at finding out how RDF data could be stored through the Apache Lucene framework in order to most efficiently retrieve RDF data from SPARQL filter queries. This was implemented by designing an index structure where the graph name, as well the subjects and objects of every triple, are stored as separate values of Lucene document fields. The predicate of a triple acts as the name of the Lucene document field that stores the object of the same triple. This way, it is easy to retrieve relevant triples from the index based on the results from the Lucene queries constructed from the SPARQL filter clauses, as one can easily look up the predicate of each triple in an RDF graph and retrieve the object connected to that predicate. Hence, the index structure makes it easy to retrieve the subject, predicate and object from any RDF graph, which is vital in order to construct the local RDF model where the general SPARQL query is executed. In terms of fulfilling the objectives of FILT, this index structure was found to be most suitable. A vital criterion in order to make FILT a compatible solution for storing, querying and retrieving RDF data through SPARQL was presented as the third success criterion of this project. The success criterion stated that all results returned from SPARQL queries should be returned in the same format as a conventional triplestore. This project accomplished this criterion, and FILT returns the results of SPARQL queries of any of the four query forms SELECT, DESCRIBE, CONSTRUCT and ASK in the exact same format as a conventional triplestore.

The third research question aimed at finding out how filter expressions of SPARQL filter queries should be re-written in order to utilize the possibilities, and cope with the restrictions, of the querying module of the Apache Lucene framework. This project has presented a working implementation of re-writing SPARQL filter queries to be compatible with the Apache Lucene query library, thus retrieving the same query results as in a conventional triplestore. The filter expressions of SPARQL filter queries were re-written by following a pre-defined algorithm for extracting the filter expression values, the filter clause type, and the triple-component that was the basis for the filtering (see section 3.2). The most challenging aspect of re-writing the SPARQL queries in FILT is the aspect of coping logical operators in the logical/numerical filter clauses. FILT copes with this by utilizing with the built-in BooleanClause.Occur operators in Lucene, namely applying the BooleanClause.Occur.MUST operator for the Boolean AND operator, and the BooleanClause.Occur.SHOULD operator for the Boolean OR operator (see section 3.3.2). This implementation works for the simple basic graph pattern SPARQL filter queries, but for large and complex queries this implementation is not fully compatible. However, there is no indication as to why this implementation should not be possible to develop further in order to be compatible with any SPARQL query. Another challenging aspect of re-writing SPARQL filter clauses in FILT includes the simplifying of numerical expressions, presented in section 3.3.1.2 (see Appendix 1: Simplifying numerical expressions in SPARQL queries in FILT for a detailed overview). FILT proved that it is possible to simplify numerical expressions in order to execute them through Lucene and retrieve the same results as executing the non-simplified numerical expressions through a conventional triplestore. Moreover, even though there are several complex aspects when it comes to re-writing SPARQL queries, the implementation of FILT shows that it is possible to re-write SPARQL queries and build Lucene queries from SPARQL filter clauses to be executed through a Lucene index in an efficient way.

The fourth research question aimed at finding out how the built-in query library of Apache Lucene could support the execution of the regex and logical/numerical expression SPARQL filter clauses. Depending on what the filter expression values in the SPARQL filter query being executed represent, as well as what the filter clause type of the filter expression is, the filter clauses are extracted and executed through Lucene differently. Examples of this are the regex filter clauses which are executed through the modified RegexQuery library in Lucene, whereas the numerical filter clauses are executed through the NumericRangeQuery libraries. If both a lower number and higher number occurred in the numerical filter clauses for the same triple-object, they are both executed through the same NumericRangeQuery with a lower number and a higher number to be matched. Numeric filtering that consists of checking if a value equals or not equals another value is executed through the RegexQuery library. If there are several regular expression filter clauses in a given SPARQL filter query, the regex filter clauses are executed through several instances of the Lucene RegexQuery class, connected through the BooleanQuery class in Lucene.

A major feature that should be implemented is the possibility of querying predicates. This is not possible with the current index structure, as the predicates themselves are the actual names of the document fields in the index (see section 3.1.1.3). This could either be solved by implementing a different index structure, or simply store all the predicate names in an external file during the index process. This data could be used during the query process to make it possible to query the predicates. It is also not possible to specify full URIs in the SPARQL queries, meaning that only URIs represented by namespaces will work in the queries. This is simply a feature that was not a high priority during the development and was not allocated enough time to finish.

FILT is currently a prototype for executing SPARQL regex and logical expression filter queries. The solution is not a standalone solution for executing SPARQL queries, but rather a general SPARQL filter query processing engine compatible with any conventional triplestore. However, due to the significant results presented in this project that highlight the efficiency of FILT compared to Joseki, future work could go in the direction of making FILT a generalized standalone solution for storing and retrieving RDF data. This would include implementing FILT to be fully compatible with any SPARQL query, thus transforming it from a hybrid architecture based on both Apache Lucene and a conventional triplestore, to a homogenous architecture where the entire querying process is executed through Apache Lucene.

References

Alkhateeb, F., Baget, J. & Euzenat, J. (2009). *Extending SPARQL with regular expression patterns (for querying RDF).* Web Semantics: Science, Services and Agents on the World Wide Web, Volume 7, Issue 2, April 2009, Pages 57-73, ISSN 1570-8268, 10.1016/j.websem.2009.02.002. Available at: <http://www.sciencedirect.com/science/article/pii/S1570826809000043>

Apache Jena ARQ (2012). *ARQ - A SPARQL Processor for Jena.* Available at: <http://incubator.apache.org/jena/documentation/larq/index.html>

Apache Jena LARQ (2012). *LARQ - adding free text searches to SPARQL.* Available at: <http://incubator.apache.org/jena/documentation/query/index.html >

Apache Lucene Query Parser Syntax (2012). *Apache Lucene - Query Parser Syntax.* Available at: <http://lucene.apache.org/core/old_versioned_docs/versions/2_9_1/queryparsersyntax.html>

Apache Lucene API (2011). *Lucene 3.5.0 API.*
Available at: <http://lucene.apache.org/core/old_versioned_docs/versions/3_5_0/api/all/index.html>

Apache Lucene Core (2011). *Apache Lucene Core.* Available at: <http://lucene.apache.org/core/>

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R. & Ives, Z. (2007). *DBpedia: A Nucleus for a Web of Open Data.* Springer Berlin / Heidelberg: Lecture Notes in Computer Science, 2007, Volume 4825/2007, Pages 722-735, DOI: 10.1007/978-3-540-76298-0_52. Available at: <http://dx.doi.org/10.1007/978-3-540-76298-0_52>

Beckett, D. (2004). *RDF/XML Syntax Specification, W3C Recommendation.* Available at: < http://www.w3.org/TR/REC-rdf-syntax/>

Beckett, D. & Berners-Lee, T. (2011). *Turtle - Terse RDF Triple Language, W3C Team Submission.* Available at: <http://www.w3.org/TeamSubmission/turtle/>

Berners-Lee, T. (2006). *Linked Data – Design Issues.* ACM Press: WC3 (2006), Volume 2009, Issue 09/20, Pages 7. DOI: 10.1145/1367497.1367760. Available at: <http://www.mendeley.com/research/design-issues-linked-data/>

Biron, P., V. & Malhotra, A. (2004). *XML Schema Part 2: Datatypes Second Edition, W3C Recommendation.* Available at: <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>

Bizer, C. & Schultz, A. (2010). *Berlin SPARQL Benchmark (BSBM) - Benchmark Rules.* Available at: <http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/BenchmarkRules/>

Bizer, C. & Schultz, A. (2009). *The Berlin SPARQL Benchmark.* In the Proceedings of the International Journal on Semantic Web and Information Systems (IJSWIS), Volume 5, Issue 2, Pages 24, DOI: 10.4018/jswis.2009040101. Available at: <http://www.igi-global.com/article/berlin-sparql-benchmark/4112>

Boag, S., Chamberlin, D., Fernández, M., F., Florescu, D., Robie, J. & Siméon, J. (2010). *XQuery 1.0: An XML Query Language (Second Edition), W3C Recommendation.* Available at: <http://www.w3.org/TR/xquery/>

Bray, T., Hollander, D., Layman, A., Tobin, R. (2006). *Namespaces in XML 1.1 (Second Edition), W3C Recommendation.* Available at: <http://www.w3.org/TR/2006/REC-xml-names11-20060816/>

Caroll, J., J., Dickingson, I., Dollin, C., Reynolds, D., Seaborne, A. & Wilkinson, K. (2004). *Jena: implementing the semantic web recommendations.* In Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW Alt. '04). ACM, New York, NY, USA, Pages 74-83. DOI=10.1145/1013367.1013381 Available at: *<http://doi.acm.org/10.1145/1013367.1013381>*

Castillo, R., Rothe, C. & Leser, U. (2010). *RDFMatView: Indexing RDF Data Using Materialized SPARQL Queries.* Proceedings of the 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2010), Volume 669, Pages 80-95. Available at: <http://ceur-ws.org/Vol-669/>

Delbru, R., Campinas, S. & Tummarello, G. (2012). *Searching Web Data: an Entity Retrieval and High-Performance Indexing Model.* Web Semantics: Science, Services and Agents on the World Wide Web, Web-Scale Semantic Information Processing, Volume 10, Pages 33–58, DOI: 10.1016/j.websem.2011.04.004. Available at: <http://www.sciencedirect.com/science/article/pii/S1570826811000230>

Delbru, R., Toupikov, N., Catasta M., Tummarello, G. (2010). *A Node Indexing Scheme for Web Entity Retrieval.* Springer Berlin/Heidelberg: The Semantic Web: Research and Applications, Lecture Notes in Computer Science, Volume 6089, Pages 240-256, DOI: 10.1007/978-3-642-13489-0_17. Available at: < http://dx.doi.org/10.1007/978-3-642-13489-0_17>

Duerst, M. & Suignard, M. (2005). *Internationalized Resource Identifiers (IRIs).* Available at: <http://tools.ietf.org/html/rfc3987>

Flether, G., H., L. & Beck, P., W. (2008). *A Role-Free Approach to Indexing Large RDF Data Sets in Secondary Memory for Efficient SPARQL Evaluation.* In the Proceedings of Computing Research Repository, abs/0811.1083. Available at: < http://arxiv.org/abs/0811.1083>

Grant, J. & Beckett, D. (2004). *RDF Test Cases.* Available at: <http://www.w3.org/TR/rdf-testcases/>

Kochut, K. & Janik, M. (2007). *SPARQLeR: Extended Sparql for Semantic Association Discovery.* Springer Berlin/Heidelberg: Lecture Notes in Computer Science: The Semantic Web: Research and Applications, Volume 4519, Pages 145-159. DOI: 10.1007/978-3-540-72667-8_12. Available at: < http://dx.doi.org/10.1007/978-3-540-72667-8_12>

Lee, J., Pham M., Lee, J., Han W., Cho, H., Yu H. & Lee J. (2011). *Processing SPARQL queries with regular expressions in RDF databases.* BMC Bioinformatics 20122, Volume 12, Suppl 2. DOI: doi:10.1186/1471-2105-12-S2-S6. Available at: < http://www.biomedcentral.com/1471-2105/12/S2/S6>

Malhotra, A., Melton, J., Walsh, N. & Kay, M. (2010). *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition), W3C Recommendation.* Available at: <http://www.w3.org/TR/xpath-functions/>

Manola, F. & Miller, E. (2004). *RDF Primer, W3C Recommendation.* Available at: <http://www.w3.org/TR/rdf-primer/>

Minack, E., Sauermann, L., Grimnes, G., Fluit, C. & Broekstra, J. (2008). *The Sesame LuceneSail: RDF Queries with Full-text Search.* NEPOMUK Technical Report 2008-1. Available at: <http://www.dfki.uni-kl.de/~sauermann/papers/Minack%2B2008.pdf>

NEPOMUK (2008). *NEPOMUK - The Social Semantic Desktop - FP6-027705.* Available at: <http://nepomuk.semanticdesktop.org/nepomuk/>

Nowack, B. (2005). *ARC: appmosphere RDF Classes for PHP Developers.* Proceedings of the SFSW 05 Workshop on Scripting for the Semantic Web . Available at: <http://ceur-ws.org/Vol-135/>

Nunamaker Jr., J. F. & Chen, M. (1990*). Systems Development in Information Systems Research.* In the Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, Volume 3, Pages 631-640. Available at: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=205401>

Oldakowski, R., Bizer, C. & Westphal, D. (2005). *RAP: RDF API for PHP.* Proceedings of the SFSW 05 Workshop on Scripting for the Semantic Web . Available at: <http://ceur-ws.org/Vol-135/>

OpenLink Software (2009). *OpenLink Virtuoso Universal Server: Documentation. RDF and Geometry.* Retrieved May 13th, 2012.
Available at: <http://docs.openlinksw.com/virtuoso/rdfsparqlgeospat.html>

Oren, E., Delbru R., Catasta, M., Cyganiak, R., Stenzhorn, H. & Tummarello, G. (2008). *Sindice.com: A Document-oriented Lookup Index for Open Linked Data.* In Proceedings of the International Journal of Metadata, Semantics and Ontologies, Volume 3, Number 1/2008, Pages 37-52, DOI: 10.1504/IJMSO.2008.021204. Available at: <http://inderscience.metapress.com/content/3518208222365647/>

Prud'hommeaux, E. & Seaborne, A. (2008). *SPARQL Query Language for RDF*. W3C working draft, 4 (January). Available at: <http://www.w3.org/TR/rdf-sparql-query/>

Rosenberg, W. & Donald, A. (1995). *Evidence based medicine: an approach to clinical problem-solving.* BMJ 1995:310:1122.

Transaction Processing Performance Council (TCP) (2010). *TPC BENCHMARK™ C. Standard Specification, Revision 5.11.* Available at: <http://www.tpc.org/tpcc/spec/tpcc_current.pdf>

U.S. Library of Congress (2010). *Codes for the Representation of Names of Languages.* Available at: <http://www.loc.gov/standards/iso639-2/php/code_list.php>

U.S. National Library of Medicine (2012). *Digoxin Oral.* Available at: <http://www.nlm.nih.gov/medlineplus/druginfo/meds/a682301.html>

U.S. National Library of Medicine (2012). *Theophylline.* Available at: <http://www.nlm.nih.gov/medlineplus/druginfo/meds/a681006.html>

Wang, H., Liu, Q., Penin, T., Fu, L., Zhang, L., Tran, T., Yu, Y. & Pan, Y. (2009). *Semplore: A scalable IR approach to search the Web of Data.* Web Semantics: Science, Services and Agents on the World Wide Web, Volume 7, Issue 3, September 2009, Pages 177-188, DOI: 10.1016/j.websem.2009.08.001. Available at: <http://www.sciencedirect.com/science/article/pii/S1570826809000262>

Wishart, D., S., Knox, C., Guo, A. C., Shrivastava, S., Hassanali, M., Stothard, P., Chang, Z. & Woolsey, J. (2006). Oxford University Press: *Nucl. Acids Res.* 34, Suppl 1, Pages D668-D672, DOI:10.1093/nar/gkj067. Available at: <http://nar.oxfordjournals.org/content/34/suppl_1/D668.short>

Appendices

Appendix 1: Simplifying numerical expressions in SPARQL queries in FILT

1 If the numerical operator is subtraction

$Rule: subtract\ the\ same\ number\ on\ both\ sides\ and\ reverse\ the\ orientation\ of\ the\ inequality\ sign$

Example:

(37.785834 - xsd:double(?lat) <= 0.040000):

37.785834 – 0.040000 - xsd:double(?lat) >= 0.040000 - 0.040000

37.745834 - xsd:double(?lat) >= 0

(xsd:double(?lat) >= 37.745834)

?lat must equal or have a higher value than 37.745834 in order for the initial expression to be true

2 If the numerical operator is addition

$Rule: subtract\ the\ same\ number\ on\ both\ sides$

Example:

(37.785834 + xsd:double(?lat) > 50)

50 - 37.785834 + xsd:double(?lat) > 50 – 50

12.214166 + xsd:double(?lat) > 0

(xsd:double(?lat) > 12.214166)

?lat must equal or have a higher value than 12.214166 in order for the initial expression to be true

3 If the numerical operator is division

Rule a: Divide by the same POSITIVE number on both sides
and reverse the orientation of the inequality sign

Rule b: Divide by the same NEGATIVE number on both sides

Example a:

(37.785834 / xsd:double(?lat) <= 0.040000):

(37.785834 / 0.040000) / xsd:double(?lat) >= 0.040000 / 0.040000

944.64585 / xsd:double(?lat) >= 0

(xsd:double(?lat) >= 944.64585)

?lat must equal or have a higher value than 944.64585 in order for the initial expression to be true

Example b:

(-37.785834 / xsd:double(?lat) <= 0.040000):

(-37.785834 / 0.040000) / xsd:double(?lat) <= 0.040000 / 0.040000

-944.64585 / xsd:double(?lat) <= 0

(xsd:double(?lat) <= -944.64585)

?lat must equal or have a higher value than 944.64585 in order for the initial expression to be true

4 If the numerical operator is multiplication

Rule a: Divide by the same POSITIVE number on both sides

Rule b: Divide by the same NEGATIVE number on both sides
and reverse the orientation of the inequality sign

Example a:

*(37.785834 * xsd:double(?lat) <= 0.040000):*

*(0.04 / 37.785834) * xsd:double(?lat) <= 0.040000 / 0.040000*

*0.0010585977803215883 * xsd:double(?lat) <= 0* *(xsd:double(?lat) <= 0.0010585977803215883)*

?lat must equal or have a lower value than 0.0010585977803215883 in order for the initial expression to be true

Example b:

*(-37.785834 * xsd:double(?lat) <= 0.040000):*

*(0.04 / -37.785834) * xsd:double(?lat) >= 0.040000 / 0.040000*

*-0.0010585977803215883 * xsd:double(?lat) >= 0*

(xsd:double(?lat) >= -0.0010585977803215883)

?lat must equal or have a lower value than 0.0010585977803215883 in order for the initial expression to be true