FILT – Filtering Indexed Lucene Triples – A SPARQL Filter Query Processing Engine–

Magnus Stuhr[1] and Csaba Veres[2]

[1] Computas AS, Norway
Magnus.Stuhr@computas.com
[2] University of Bergen, Bergen, Norway
Csaba.Veres@infomedia.uib.no

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. 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. This paper presents a SPARQL filter query-processing engine for conventional triplestores called FILT (Filtering Indexed Lucene Triples), built on top of the Apache Lucene framework for storing and retrieving indexed documents, compatible with unmodified SPARQL queries. The objective of FILT was to decrease the query execution time of SPARQL filter queries. This aspect 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.

Keywords: RDF full-text search, SPARQL filter queries, SPARQL regex filtering, SPARQL numerical filtering, RDF data indexing, Lucene.

1 Introduction

RDF (Resource Description Framework) is a language for describing things or entities on the World Wide Web [8]. 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. 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 [13]. 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.

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 and numerical values stored as literals in most RDF data sets. 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, also offering 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. Examples of SPARQL clauses are filtering string values, regular expressions, logical expressions and language metadata. 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. As SPARQL filter queries can discover data relationships that general SPARQL queries cannot, they play an important role in retrieving RDF data. However, because 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. [9] 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 paper presents a technique for optimizing the query-execution times of SPARQL filter queries. A prototype solution called FILT (Filtering Indexed Lucene Triples) has been built in order to show that a general SPARQL filter query processor can decrease the query-execution time of SPARQL filter queries, thus enhancing the value of integrating full-text searches with the SPARQL query language. The paper is divided into six sections apart from the introduction: section 2 presents the implementation and features of FILT, section 3 presents previous related work, section 4 presents the framework for evaluating FILT through a benchmark test, section 5 presents the results of the benchmark test, section 6 discuss the results, and finally section 7 presents conclusions and further work.

2 Implementation of FILT with Apache Lucene

FILT is a SPARQL filter-processing engine and enables storing and querying of RDF data through the Apache Lucene framework [3]. It is supports unmodified SPARQL queries, meaning that users do not have to re-write their SPARQL queries in order to execute them. The main purpose of FILT 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 RDF 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 are 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 execute properly. The architecture of FILT is shown in Figure 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 that 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 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. 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 RDF model 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 is returned to the user that issued the SPARQL query.

As mentioned, FILT is built on top of the Apache Lucene framework. 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. Lucene is a vital part of storing and querying data in FILT. A Lucene index contains a set of documents that contain 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 that can be indexed so that terms in the field can be used to retrieve the document through Lucene queries. The index structure in FILT is based on a dynamic index structure that

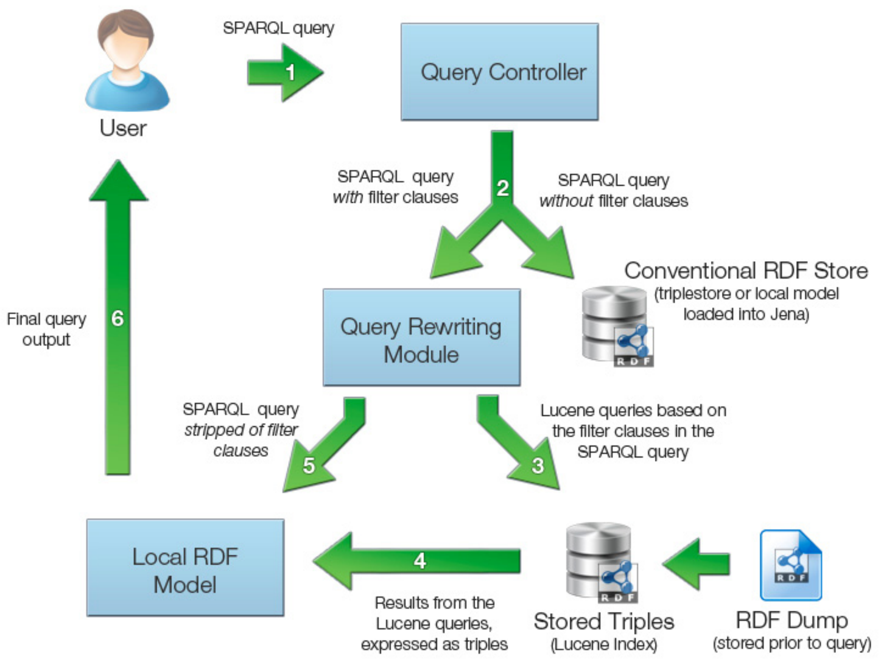

Fig. 1. The architecture of FILT

contains a default "graph" and "subject" fields, which contain the RDF graph locations (paths to data set files) and the subject URI of the data entity being indexed. Apart from this, the index structure is a dynamic structure that names each field in a document by its 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. This makes it easy to query the index by specifying predicate names for the field names in the Lucene queries. 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>)
   }
}
```

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 run through the local model or the SPARQL endpoint specified by the data set owner. FILT has mainly focused on implementing compatibility with SPARQL regex filter clauses and SPARQL logical/numerical expression filter clauses. 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. To illustrate how FILT deals with the aspect of number filtering, 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)}.* The 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).*$).* 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 numeric operators. 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 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 have been divided into distinct Lucene queries, these different Lucene queries will be joined as one large query and finally executed over the index.

3 Previous Work

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 [12]. SIREn is a semantic information retrieval engine plugin to Lucene [7], and is the search engine that Sindice is based on. SIREn includes a node-based indexing scheme for semi-structured data, based on the Entity-Attribute value model [6]. 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.

SEMPLORE [14] 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. In addition, FILT is a database solution opposed to SEMPLORE, which is mainly a web solution.

Castillo et al. [5] 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. [5] 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.

There exist several solutions trying to implement efficient full-text searches through the SPARQL query language. Apache Jena LARQ [2] is a querying solution based on Lucene and the Jena SPARQL query engine Apache Jena ARQ [1]. NEPOMUK [10] 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. In addition, 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. [9] 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. 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. In addition, 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 [11]. 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 simply execute queries of the standard SPARQL syntax.

4 Benchmark Evaluation

In this project, an extensive benchmark evaluation of FILT has been performed. The objective of the benchmark test was to compare the features of FILT to the Joseki triplestore by evaluating several metrics regarding the speed of query execution. 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. 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 was tested. 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 will be 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 DrugBank data set can be downloaded from: http://dl.dropbox.com/u/21236338/drugbank.zip. The Geographical Coordinates of DBpedia data set can be downloaded from: http://downloads.dbpedia.org/ 3.7/en/geo_coordinates_en.nt.bz2.

The metrics of this benchmark evaluation are based on the performance metrics specified by Bizer & Shultz [4]. The metrics are "Milliseconds per Query (MSpQ)", "Average Query Execution Time (aQET)", "Overall Runtime (oaRT)" and "Average Query Execution Time over all Queries (aQEToA)". However, the benchmark evaluation in this paper will only evaluate and present the aQET. The aQET will be 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. The query mixes of both the regex use-case and the numerical filtering use-case contained 24 queries; six queries of each SPARQL query form (SELECT, DESCRIBE, CONSTRUCT and ASK). This way, the performance of all the query forms isolated could be analyzed. The query mixes were executed three times for each data set sizes. Prior to each execution of the query mixes, the data sets were re-loaded along with executing a warm-up query-mix.

5 Results

This section will refer to each of the data set sizes of the DrugBank and Geographical Coordinates data sets 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 “*” where the probability number is less than 0.05, “**” where the probability number is less than 0.01, and “***” where the probability is less than 0.001.

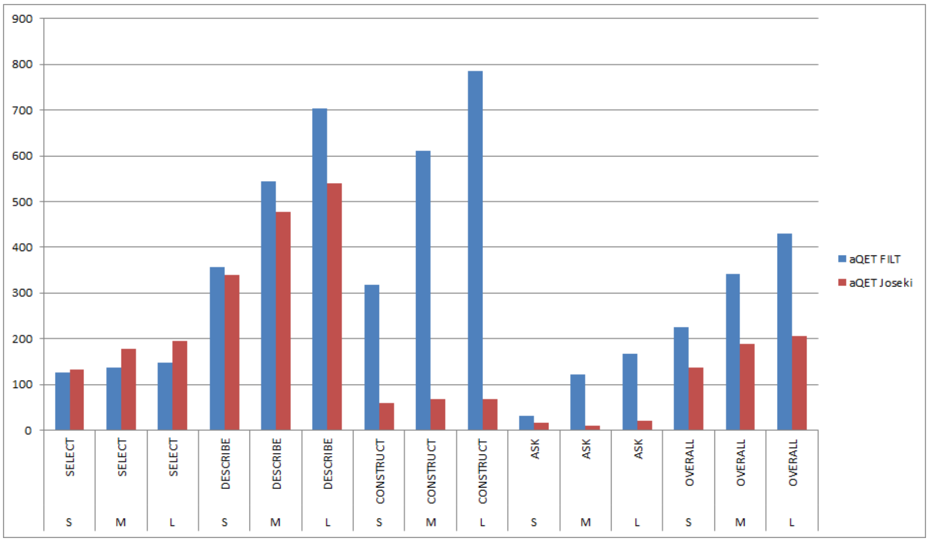

Fig. 2. The overall benchmark results of the DrugBank regular expression filtering use-case

The overall results of the DrugBank regular expression filtering use-case are shown in Figure 2. 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. 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. The probability numbers showed 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 $p < 0.10$. Further, the chart 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. The statistics made it evident 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 $p < 0.01$. The results also 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. The statistics made it 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 $p < 0.001$.

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 queryieswere 145 milliseconds. The statistics made it evident that that the data set size (Size) is not a significant factor when executing the ASK query 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 $p = 0.076$. The overall aQET of all queries in the query mix shows 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. The statistics made it 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 $p < 0.001$.

To summarize the SPARQL regex 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.

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 3 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. The statistics made it evident that the data set size (Size) is a significant factor when

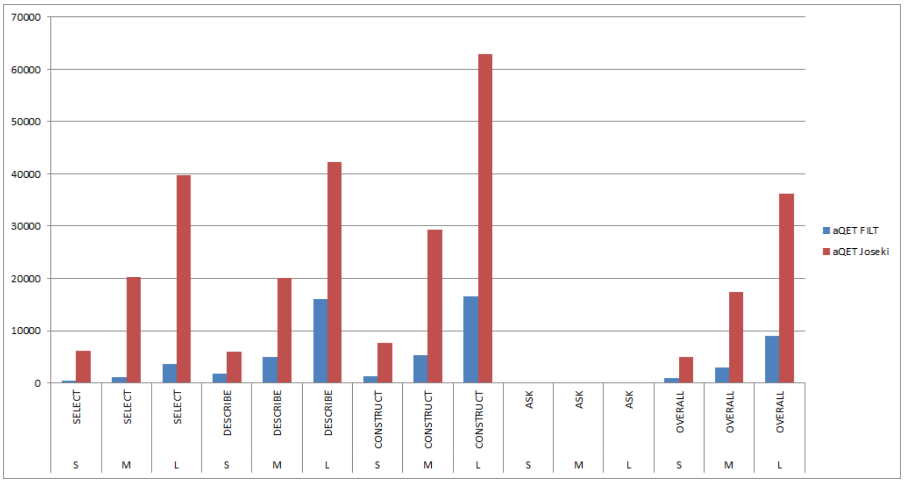

Fig. 3. The overall benchmark results of the Geographical Coordinates numerical/logical filtering use-case

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.

Further, the chart shows 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). The statistics made it 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.

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). The statistics made it 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. 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. The statistics shows that the data set size (Size) is a significant factor when executing the ASK query 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 $p < 0.001$. The results show the overall aQET of all queries in the query mix. The statistics made it clear 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, $p < 0.001$.

To summarize the SPARQL numerical/logical filter query use-case, FILT outperforms Joseki to a great extent for 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.

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. It is worth mentioning that even though Joseki performs better than FILT for the CONSTRUCT, DESCRIBE and ASK query forms in the SPARQL regex filter query use-case; 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, certain aspects of how FILT returns query results 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. 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.

The fact that Joseki struggles largely 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. Additionally, if the results of both use-cases were combined into one huge result set, FILT would outperform Joseki to a great extent. This is because even though FILT performs slightly slower than Joseki in the SPARQL regex 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.

7 Conclusions and Future Work

This paper has demonstrated the practical advantages of using a text-indexing platform in conjunction with a regular triplestore, for executing certain kinds of SPARQL queries. Our implementation of FILT, based on Lucene, demonstrated that in the most successful cases, FILT returned results 46 seconds faster than Joseki. In usability terms, a18 second response from FILT is far more acceptable than a 64-second response from Joseki. The aim now is to implement FILT as a general architecture that can be deployed by any triplestore maintainer. The advantage of our approach is that it is agnostic about the companion triplestore, and does not require any special syntax. In other words, it can be transparently deployed alongside any triplestore.

A number of outstanding issues need to be resolved. First, we need to solve the puzzling limitations in CONSTRUCT and DESCRIBE regex filter queries, as well as ASK queries of both regex and numerical SPARQL filter queries. Second, we need to include more rewrite rules to cope with the full range of FILTER queries. Finally, we need to ensure that the solution is scalable to any required implementation. Once these issues are resolved, FILT will be distributed as a simple package that will handle the indexing of RDF data in the triplestore, and be deployed as a seamless layer that passes non-FILTER queries onto the regular triplestore, but executes FILTER queries through its own speedy execution engine.

References

[1] Apache Jena ARQ, ARQ - A SPARQL Processor for Jena (2012), `http://incubator.apache.org/jena/documentation/larq/index.html`

[2] Apache Jena LARQ (2012), LARQ - adding free text searches to SPARQL, http://incubator.apache.org/jena/documentation/query/index.html

[3] Apache Lucene Core, Apache Lucene Core (2011), http://lucene.apache.org/core/

[4] Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. The Proceedings of the International Journal on Semantic Web and Information Systems (IJSWIS) 5(2), 24 (2009), http://www.igi-global.com/article/berlin-sparql-benchmark/4112, doi:10.4018/jswis.2009040101

[5] Castillo, R., Rothe, C., Leser, U.: RDFMatView: Indexing RDF Data Using Materialized SPARQL Queries. In: Proceedings of the 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2010), vol. 669, pp. 80–95 (2010), http://ceur-ws.org/Vol-669

[6] Delbru, R., Campinas, S., Tummarello, G.: 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 10, 33–58 (2012), http://www.sciencedirect.com/science/article/pii/S1570826811000230, doi:10.1016/j.websem.2011.04.004

[7] Delbru, R., Toupikov, N., Catasta, M., Tummarello, G.: A Node Indexing Scheme for Web Entity Retrieval. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010, Part II. LNCS, vol. 6089, pp. 240–256. Springer, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-13489-0_17, doi:10.1007/978-3-642-13489-0_17.

[8] Manola, F., Miller, E.: RDF Primer, W3C Recommendation (2004), http://www.w3.org/TR/rdf-primer/

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

[10] NEPOMUK, NEPOMUK - The Social Semantic Desktop - FP6-027705 (2008), http://nepomuk.semanticdesktop.org/nepomuk/

[11] OpenLink Software, OpenLink Virtuoso Universal Server: Documentation. RDF and Geometry (2009), http://docs.openlinksw.com/virtuoso/rdfsparqlgeospat.html (retrieved May 13, 2012)

[12] Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice.com: A Document-oriented Lookup Index for Open Linked Data. Proceedings of the International Journal of Metadata, Semantics and Ontologies 3(1/2008), 37–52 (2008), http://inderscience.metapress.com/content/3518208222365647, doi:10.1504/IJMSO.2008.021204

[13] Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C working draft, 4 (January 2008), http://www.w3.org/TR/rdf-sparql-query

[14] Wang, H., Liu, Q., Penin, T., Fu, L., Zhang, L., Tran, T., Yu, Y., Pan, Y.: Semplore: A scalable IR approach to search the Web of Data. Web Semantics: Science, Services and Agents on the World Wide Web 7(3), 177–188 (2009), http://www.sciencedirect.com/science/article/pii/S1570826809000262, doi:10.1016/j.websem.2009.08.001