UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

An Investigation into the Performance of Regular Expressions within SPARQL Query Language

by

Saud Aljaloud

Thesis for the degree of Doctor of Philosophy

January 2019

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

AN INVESTIGATION INTO THE PERFORMANCE OF REGULAR EXPRESSIONS WITHIN SPARQL QUERY LANGUAGE

by Saud Aljaloud

SPARQL has not simply been the standard querying language for the Resource Description Framework (RDF) within the Semantic Web, but it has also gradually become one of the main querying languages for the graph model, in general. To be able to process SPARQL in a more efficient manner, an RDF store (as a DBMS) has to be used. However, SPARQL faces huge performance challenges for various reasons: the high flexibility of RDF model, the fact that the SPARQL standardisation does not always focus on the performance side, or the immaturity of RDF and SPARQL in comparison to some other models such as SQL.

One of SPARQL features is the ability to search through literals/strings by using a Regular Expression (Regex) filter. This adds a very handy and expressive utility, which allows users to search through strings or filter certain URIs. However, Regex is computationally expensive as well as resource intensive in that, for example, data has to be loaded into the memory.

This thesis aims to investigate the performance of Regex within SPARQL. Firstly, we propose an analysis of the way people use Regex within SPARQL by looking at a huge log of queries made available provided by various RDF store providers. The analysis indicates various use cases in which their performance can be made more efficient.

There is very little in the literature to adequately test the performance of Regex within SPARQL. We also propose the first Regex-Specific benchmark, named (BSBMstr) to be applied to the area of SPARQL. BSBMstr shows how various Regex features affect the overall performance of the SPARQL queries. BSBMstr also reports its results on seven known RDF stores.

SPARQL benchmarks, in general, have been a major field that attracts much research in the area of the Semantic Web. Nevertheless, many have argued that there are still issues in their design or simulation of real-world scenarios. This thesis also proposes a generic

SPARQL benchmark, named CBSBench which introduces a new design of benchmarks. Unlike other benchmarks, CBSBench measures the performance of clusters rather than fixed queries. The usage of clusters also provides a stress test on RDF stores, because of the diversity of queries within each cluster. the CBSBench results are also reported on five different RDF stores.

Finally, the thesis introduces (Reg|Ind)ex which is a Regex index data structure that is based on a tri-grams inverted index. This index aims to reduce the result sets to be scanned to match a Regex filter within SPARQL. The proposal has been evaluated by two different Regex-specific benchmarks and implemented on top of two RDF stores. (Reg|Ind)ex produces a smaller index size compared to previous work, while still being able to produce results faster than the original implementations by up to an order of magnitude.

In general, the thesis provide a general guidelines that can be followed by developers to investigate similar features within a given DBMS. The investigation mainly relies on real-world usage by analysing how people are using these features. From that analysis, developers can construct queries and features alongside our proposed benchmarks to run tests on their chosen subject. The thesis also discusses various ideas and techniques that can be used to enhance the performance of DBMSs.

Contents

List of Figures

List of Tables

Listings

Glossary

API A generic term in computer science that provides an access to some sort of resources. API stands for Application Programming Interface.. 31

BDMS DataBase Management System.. 1, 2, 4, 6, 9–12, 16, 17, 23, 24, 26, 29, 36, 38, 52, 59, 69, 79, 90, 111

BGP In SPARQL, the main part of the query which describes the graph pattern is refer to as Basic Graph Pattern BGP.. 33, 39, 44, 47, 52, 54, 72, 98

BSBM Both BSBM and BIBMis being used interchangeably to refer to the Berlin SPARQL Benchmark.. ix, xiii, 3, 4, 6, 7, 29, 32, 51–54, 56–62, 66, 67, 69–72, 77, 80, 100–102, 105, 109, 112

FTS A term that refers to Full-Text Search (FTS) facility that is usually integrated with a particular system.. 24, 31, 34, 40, 44, 46–48, 74, 90, 92

HD A term that refers to Hard Disk.. 11–14

QMPH a measurement unit that stands for Query Mix Per Hour. The greater number, the better performance a query is.. ix, x, 51, 54, 56–59, 63, 65, 66, 76–78, 81, 82, 84, 86, 87, 108

QPS a measurement unit that stands for Query Per Second. The greater number, the better performance a query is.. ix, x, 3, 51, 55, 57, 58, 60, 63–65, 76–78, 81–83, 103

RDF The Resource Description Framework (RDF) is a standard model for data interchange on the Web. 1, 9, 39, 51, 69, 90, 111

Regex Stands for Regular Expressions and sometimes refer to as Regexes. 1, 9, 39, 51

SPARQL is a programmable machine that receives input, stores and manipulates data, and provides output in a useful format. 1, 9, 39, 51, 69, 89, 111

SQL SQL is a domain-specific language used in programming and designed for managing data held in a relational database management system, or for stream processing in a relational data stream management system.. 10, 15, 24, 29, 34, 38, 79, 90, 111

SSD An SSD (solid-state drive) is a type of non-volatile storage media that stores persistent data on solid-state flash memory.. 12, 17

STR An acronym for String.. 47

TPC The Transaction Processing Performance Council. 17

URI A URI (Uniform Resource Identifier) is a string that refers to a resource. The most common are URLs, which identify the resource by giving its location on the Web.. 25, 27, 47, 93, 107

XML Extensible Mark-up Language (XML) is a mark-up language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.. 1, 33, 45, 54, 91

Declaration of Authorship

I, Saud Aljaloud , declare that the thesis entitled *An Investigation into the Performance of Regular Expressions within SPARQL Query Language* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- none of this work has been published before submission

Signed:.....

Date:.....

Acknowledgements

I am mainly thankful for my employer and sponsor, University of Ha'il for the trust to invest in me during all those years of sponsorship. They have been very supportive and cooperative in all aspects. I would also like to thank both Nick Gibbins and Tim Shown for their guidance as supervisors. During my study in the lab, I was very grateful to meet Manuel Leon Urrutia, Ramine Tinati and Markus Luczak-Roesch, they provided their experience and knowledge on working together on various academic topics.

Chapter 1

Introduction

The web today has an enormous amount of information, which makes it relatively hard to answer specific questions, as in searching, exchanging or even harder to reason about (such as: deducing new information based on existing ones). Data on the web is mainly treated as a set of documents, that are referred to with URLs; making it simpler for people to adapt and use such a system.

The Semantic Web (Berners-Lee et al., 2001) or Web of Data (Bizer et al., 2009a) is the area for creating, publishing or interlinking data on the web for the purpose of machines being able not to just display data, but also to automate, integrate or reuse the data within different applications. Things on the web should be pointed at with URIs; they may also be defined by other people using other providers. This requires a unified model that is openly accessible. In the same way as web services work, Linked Data providers should also provide end-points that are used by either individuals or machines. End-points should be used within a standard querying language (SPARQL) (Prud'Hommeaux and Seaborne, 2008) and its protocol, which should also be expressible using the Resource Description Framework data model (RDF) (Klyne et al., 2004). As the amount of data is expected to be huge within its providers, a Database Management System (BDMS) is also required to manage these data. These BDMSs have some differences from other BDMS models, such as the relational model (Neumann and Weikum, 2008), since RDF triples are verbose, in the sense that each triple has to be represented by three, or sometimes four components.

Regular expressions (Regex) are also a part of the SPARQL specification. They are used for searching through literal strings or other resources. Regex have been studied extensively in the context of theoretical computer science (Gruber and Holzer, 2014). They have a clear definition through different languages such as: XML Schema[1], XPath or

[1] `http://www.w3.org/TR/xmlschema-2/#regexs`

XQuery[2]. They are also powerful in their expressivity, which not only can describe regular languages but also some other irregular ones, such as the Regex engines embedded within modern languages as in Perl.

From a practical point of view, people who may not be familiar with how to write optimised queries, may query the data in an inefficient way. This makes query optimisations within SPARQL even more challenging than with other models, which do not provide direct access to their BDMS engine and, therefore, can choose more optimised queries by their BDMS's administrator, for example.

1.1 Problem Statement

BDMSs are supposed to be able to respond quickly to queries about the data they provide. One way of achieving this requirement would be by reducing the size of the data to be processed before responding to queries, for example, by using BDMS data structures in which only a subset of the data will be processed. For instance, a query about a user with a specific first name does not usually require the BDMS to scan all users' data.

Semantic Web technologies such as RDF and SPARQL, when wrapped within an RDF store, have even more challenges than the traditional relational ones. Firstly, they are still immature in comparison to their competitors. For example, SPARQL 1.1 (Seaborne et al., 2013) is still adding essential features, such as "Property Paths", which was not part of the previous SPARQL standardisation. Secondly, the RDF data model has a very generic schema or is even sometimes referred to as "schema-free" (Aluc et al., 2014). RDF data and/or queries could then be unpredictable; different RDF datasets or log queries can be very different from each other. RDF stores have to expect a variety of types of data and queries, which makes their job even more difficult.

Given the challenges of BDMSs in general, as well as the specific burden of RDF stores, there is also the problem of Regular Expressions. These are highly expressive in describing string patterns in the area of string matching. They are, however, computationally expensive and cause high memory usage. Most BDMSs, including RDF stores, have included Regular Expressions to be used within their query languages to allow for querying string patterns. However, because of the high expressivity of Regex, they are not always used by people in an efficient manner.

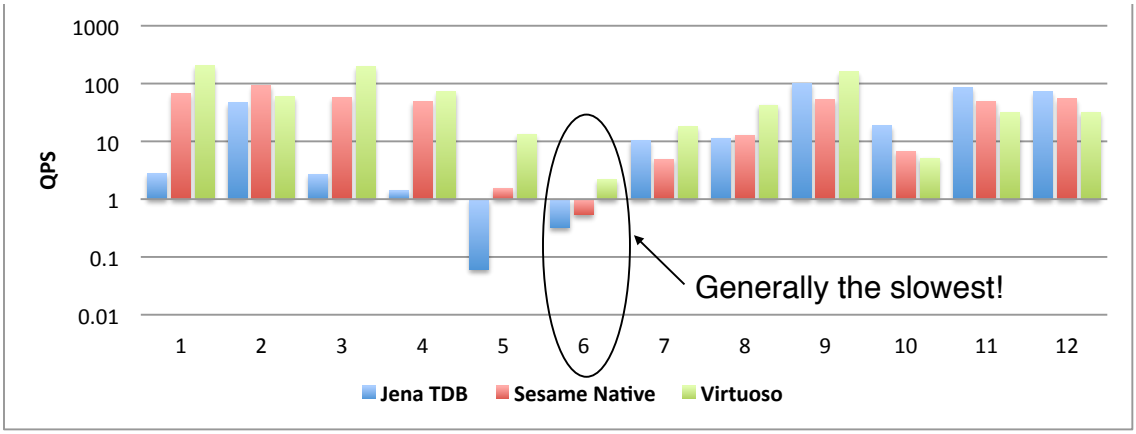


Figure 1.1: BSBM result of running the benchmark on RDF stores with **25** million triples (Bizer and Schultz, 2009).

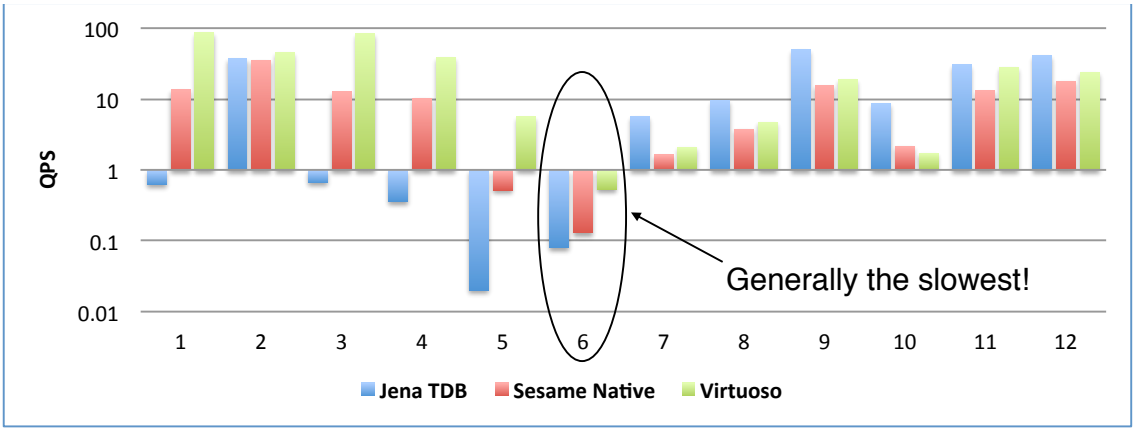


Figure 1.2: BSBM result of running the benchmark on RDF stores with **100** million triples (Bizer and Schultz, 2009).

1.2 Motivation

The well-known Berlin SPARQL benchmark (BSBM) has published its results about the performance of different RDF stores. It uses a set of 12 queries in which it assumes that these queries reflect a generic testing of RDF stores (Bizer and Schultz, 2009). Both Figures 1.1 and 1.2 show the original BSBM's results of running the benchmark on three different RDF stores using two workloads (25 and 100 million triples) from the BSBM dataset[3]. The two figures show that there are two queries (5 and 6) which have a noticeably slower performance than the others, given that for a particular Query Per Second (QPS); the greater the QPS, the better it performs. Query (5) has much more features such as: complex *FILTER*, *LIMIT*, *ORDER BY* and *DISTINCT* modifiers as well as seven triple patterns. Query (6) (presented in 1.1), however, has a simpler structure, with a single *REGEX* filter and two triple patterns. However, Query (6) is generally

[2] `http://www.w3.org/TR/xpath-functions/#regex-syntax`

[3] `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/V2/results/index.html`

slower than (6) and subsequently is the slowest of all the BSBM queries. Moreover, Figure 1.2 indicates that Query (6) is becoming relatively much slower than (5) as the data gets bigger, predicting a scalability issue within the query. BSBM has been updating their benchmark results ever since, but they removed this query as it "would become very expensive in the 1G and larger tests, so its performance would dominate the results"[4]. Apart from the initial BSBM, other benchmarks do not seem to be in favour of testing Regex. This is also probably for the same reason that BSBM removed Query (6).

```
SELECT ?product ?label
WHERE {
    ?product rdfs:label ?label .
    ?product rdf:type bsbm:Product .
        FILTER regex(?label, "%word1%")
}
```

Listing 1.1: Query 6 within the BSBM Benchmark.

1.3 Scope

The range of this study can be very wide and, subsequently, harder to comprehend or to reach accurate conclusions. To investigate the problem of Regex performance within SPARQL, we assume that an RDF store is being used as a BDMS. Such a BDMS is processing data that cannot/is not intended to fit into the main memory. It also assumed that the BDMS is run on a single machine, in contrast to distributed/clustered nodes.

The topic of Regular Expressions is also a broad area; we study the problem of exact matching of Regex, while other usages exist, such as approximate approaches. We also assume that Regex are being used for natural language purposes, such as the English language. Those languages tend to have larger alphabets, unlike other representations, such as DNA/protein sequences or binary digits.

1.4 Research Questions

This section specifies the research questions that will try to be answered throughout this thesis. They are as follows:

1. What are the trends/patterns that can be identified in people's usage of Regex within SPARQL? (Chapter 3)

[4] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/index.html

2. What are the factors that empirically affect the performance of Regex within SPARQL? (Chapter 4)
3. What are the issues with the current generic SPARQL benchmarks? (Chapter 5)
4. What optimisations can be done to increase the performance of Regex within SPARQL? (Chapter 6)

1.5 Aims and Objectives

The aim of this thesis is to optimise the performance of Regex queries within SPARQL query language. To be able to achieve this aim, we set up a number of objectives that we intend to follow:

- **Objective 1:** Identify Regex patterns within SPARQL that people often use by analysing log files posed on different SPARQL end-points from various domains. (Chapter 3)
- **Objective 2:** Design a Regex-specific benchmark that enables developers to test RDF stores on their performance against different Regex patterns. (Chapter 4)
- **Objective 3:** Design a generic SPARQL benchmark that is based on real-world set of queries grouped into clusters according to the syntactic features the queries share. (Chapter 5)
- **Objective 4:** Design a Regex-specific index that can be embedded with given RDF stores to improve the performance of Regex filters within SPARQL queries, while maintaining a relatively small index size. (Chapter 6)

1.6 Contributions

The research described in this thesis makes the following key contributions to addressing the issue of expensive evaluation of Regular Expressions in the context of SPARQL. Noting that each contribution is linked with specific research question, aim as well as a specific chapter.

- An analysis of how people are actually using Regex within SPARQL. This is done by analysing large real-world SPARQL queries that have been posed by users on different SPARQL end-points, which have been gathered from different domains. This analysis aims to validate Aim 1.5. Analysing Regular Expression patterns has not been discussed previously. The result of this analysis has affected other

contributions in this thesis, such as those of Aims 1.5 and 1.5. This contribution is discussed in details in Chapter 3. The result of this study has also been published:

Aljaloud, Saud, Luczak-Rsch, Markus, Chown, Tim and Gibbins, Nicholas (2014) Get All, Filter Details - On the Use of Regular Expressions in SPARQL Queries. In, 4th Workshop on Usage Analysis and the Web of Data in ESWC'2014, Crete, GR, 25 May 2014.

- This thesis proposes BSBMstr, a SPARQL Regex benchmark that extends the Berlin SPARQL Benchmark BSBM. This proposal is related to Aim 1.5. Not only Regular Expressions lack generic benchmarks in the context of BDMSs, but also SPARQL benchmarks have had very little testing on Regex. This is the first SPARQL-specific benchmark for the performance of Regex. Testing the performance of Regex within different RDF stores can reveal some insights as to why there might be some differences in the performance. This will lead to a better understanding of what affects the performance of Regex within SPARQL. A more detailed description can be found in Chapter 4.

- This research presents a generic SPARQL benchmark (CBSBench) that is based on real-world data and queries. This approach is related to Aim 1.5 The benchmark, unlike others, measures the performance of different clusters instead of queries. Each cluster includes a set of queries that are grouped, based on a set of SPARQL features. CBSBench performs a stress test on RDF stores by firing different queries from each cluster with each run. This strategy puts a greater challenge on RDF stores' caching mechanisms. CBSBench also reports its results on five different known RDF stores. Refer to Chapter 5 for more details.

- This research also proposes (Reg|Ind)ex, a Regex index that reduces the intermediate results to be scanned within the Regex engine. The proposal outperforms the native method of evaluating a Regex query by an order of magnitude when applied to the BSBMstr query set. This proposal relates to Aim 1.5. In general, the notion of building index structures for Regular Expressions is not new. Within SPARQL, we are only aware of RegScan (Lee et al., 2011). Our proposal can uses a tri-gram index, which processes the index faster. It also can be embedded within RDF stores more easily. The proposal is described in Chapter 6.

.

1.7 Thesis Structure

This thesis is structured as follows:

- Chapter 2 presents a literature review of the three main areas with which this thesis is concerned. Firstly, Database Management Systems are discussed within the dominant Relational Model, in terms of storing, indexing and querying. This provides the essential understanding of why different techniques or theories, such as `B+Tree` or the relational algebra, are considered. Then, Regular Expressions are discussed within the umbrella of string matching problems. Finally, the SPARQL query language is discussed as part of the knowledge of the Semantic Web. Regular expressions, as part of the SPARQL specifications, are also explored in terms of optimisations at both the physical and logical levels.

- Chapter 3 builds up the initial step towards the optimisation of Regular Expressions within SPARQL. Since Regular Expressions are highly expressive, we start by analysing a large set of real-world query logs that have been posed by real users on different SPARQL end-points. This shows the trends of how people are actually using Regex within SPARQL. It analyses Regex in terms of operators/features usage, triple pattern positions and their relation to a full-text searching.

- Chapter 4 proposes BSBMstr, a benchmark for Regex queries within SPARQL. It extends BSBM, a known generic SPARQL benchmark, and proposes a set of Regex queries, which not only measure the performance of Regex within SPARQL against different RDF stores but also identify what makes Regex queries slow within SPARQL.

- Chapter 5 addresses the issues faced by the most current SPARQL benchmark and introduces a generic SPARQL benchmark the we, referred to as CBSBench. It is based on real-world data and queries to build a set of cluster queries to be tested against RDF stores.

- Chapter 6 proposes a Regex Index named (Reg|Ind)ex, in which we build an inverted tri-gram index on the literal values of the RDF data, which reduces the intermediate result to be scanned. We evaluate both (Reg|Ind)ex and the native method within BSBMstr. Our proposal outperforms the native method by an order of magnitude in most of the queries in BSBMstr.

- Chapter 7, finally, presents this thesis's conclusion and offer suggestions for future work that can be carried out, based on the work that has been done in this thesis.

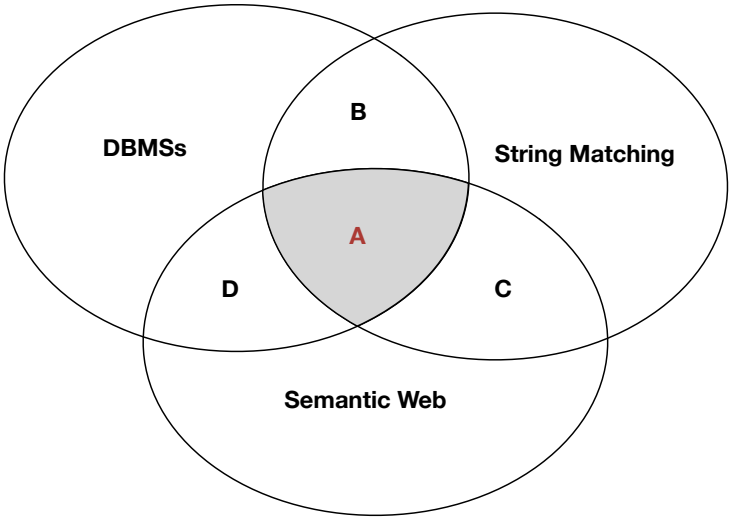
Following the above structure, there are also a number of appendices:

- Appendix A provides snippets from the tool used to parse and generate statistics and how people are actually using Regex.
- Appendix B presents the queries and their results which have been used with the BSBMstr benchmark.
- Appendix C also presents snippets about queries, clustering logs and results that have been used within the CBSBench benchmark.
- Appendix D also describes the queries being used with (Reg|Ind)ex, alongside small code samples about building and querying the Regex index.

Chapter 2

Literature Review

To investigate the performance of Regex within SPARQL, there are three main research areas depicted in Figure 2.1 which have to be reviewed: Database Management Systems (BDMS), the String matching problem and the Semantic Web. The standard query language for the RDF data model is SPARQL, which may be described in terms of relational algebra (Cyganiak, 2005). Then, Regular expressions are part of a bigger problem, known as the string matching problem. Finally, both SPARQL and RDF are



B: Intersection between DBMS and String Matching which covers various topics i.e., exact matching within SQL.

C: Intersection between String Matching and Semantic Web i.e., keyword search within Semantic Web.

D: Intersection between DBMS and Semantic Web i.e., building an RDF database that supports SPARQL.

A: This is the focus of this research area - Processing Regex within SPARQL.

Figure 2.1: A Venn diagram that shows the main areas of this research.

part of the Semantic Web area. This chapter provides a review for each of these areas to narrow down the problem this thesis aims to solve.

2.1 Database Management Systems

Database Management Systems (BDMSs) evolved to provide a separation of most of the internal processes that can be abstracted away from users (Coronel et al., 2009). This can be done by providing a package of software that can take care of different operations, such as data manipulation (storage, querying), concurrency control, transaction properties (ACID) and recovery, etc. For the scope of this thesis, the author will discuss only the parts of these issues that mainly concern the querying process, such as storing and indexing. There have been different BDMS models proposed since the emergence of the dominant relational model.

2.1.1 Relational Data Model

The Relational Data Model is the predominant model used for databases. It started in the banking domain, when Ted Codd published his paper (Codd, 1970). The main idea of the relational model is its data-independence, that is to say, there should be a higher abstraction of data away from its physical representation. This is done through the application of set theory and predicate logic to express how entities are related to each other. This separation between the logical and physical is important since stakeholders do not need to understand how data is physically represented on disks when querying data. Moreover, this provides a generic model that can be applied to different domains that need such data manipulations. The Structured Query Language (SQL) has become the de-facto standard for querying relational databases.

Relational databases comprise a number of *Relations* (Tables), which are sets of *Tuples* (Rows) containing values *Attribute*. The terms *Relation*, *Tuple* and *Attribute* have been coined with the relational model. The relational model requires a rigid pre-defined schema that explicitly describes the structure and constraints of the relations. This provides faster queries since these metadata are presented prior to query evaluation. This metadata is one of the strongest points of the relational model since, for a given query, it is possible to automatically generate different plans in order to choose the least costly one. However, a disadvantage of such a rigid schema is the integration with other systems, which requires either some changes in the schema to be matched with others, which is usually a costly operation, or by adding a different layer to establish a unified schema, which can result in a performance overhead.

Normalisation is an important characteristic of the Relational Model as well, which was firstly referred to as a “Consistency and redundancy” issue in Codd’s paper. For

example, instead of denoting the department name of an employee in each tuple, a relationship has to be established, such as *department names*, which then are linked to the *Employees table*. Mainly, this helps in reducing the redundancy, by only referring to department names using their primary key instead of a long-length string, and increases the consistency of data. For example, in the case of changing/deleting a department, it will only require one modification to the original record. This is also referred to as the First Normal Form (1NF) in (Garcia-Molina et al., 2009).

The operations that can be performed on a database can be decomposed into three main commands:
Select (σ): This is a unary operation that restricts the query to being applied against only tuples satisfying a condition.
Project (Π): This is also a unary operation that restricts the query to being applied against only certain attribute(s)
Join ($\bowtie$): This is, however, a binary operation that combines data against their relationship(s) between two or more relations.

2.1.2 Data Storage and Indexing

One of the important aspects of data representation is how the data is going to be physically stored on machines. This affects a number of factors, such as lookup performance, writing performance and space usage. In this thesis, the focus will be on storing data that exceeds the main-memory space and which has to be stored on secondary storage, typically hard disks (HDs).

Storing data on Random Access Memory (RAM) is relatively a faster operation (throughput in order of 10GB per second) due to the fact that accessing a page, either for reading or writing, is done in a constant time. However, there are two main issues with RAMs. Firstly, the price of RAM is higher than the price of HD, given the same size unit. Secondly, RAM is volatile, which causes data being stored in RAM to lose persistence. BDMSs require data to be reconstructed into RAM. Nevertheless, RAMs are still used for some other BDMSs purposes, such as caching, processed data and metadata of relations.

On the other hand, HDs, as a secondary storage, have different properties. HDs are much cheaper than RAM given the same size unit; they provide a relatively huge space that can accommodate databases in the order of high gigabytes on a single machine. Data is stored on small units called *blocks* that result from intersecting between sectors and tracks. However, the major issue is that the access time of HDs is relatively not as quick as RAM. The latency of HDs is caused by the combination of three factors: rotation of the platter, the seek time of the head and the transfer time of data to RAM. Therefore,

when exchanging data between external and internal storages, the input and output communication (I/O) will be in a bottleneck. For this reason, it is always intended to reduce the amount of access to the external memory as much as possible (Garcia-Molina et al., 2009).

Although rotational disks are the dominant mechanism so far, Solid-State Disk/Drive (SSD) have become used increasingly. SSDs have no rotating disks or other mechanical components, making them perform random reads 100× faster than rotational media (Tsirogiannis et al., 2009). They are also being praised for their low power consumptions, their light weight and high data bandwidth. SSDs, however, are known to cost more in price in comparison to rotational disks with the same size unit. They also have low write speed and similar sequential reads to their competitors. It has been argued that SSDs have an effect on the way we conceptualise storing and querying BDMSs. For example, as BDMSs use sequential reads heavily, SSDs are known to have similar performance with rotational disks. Others argue that there are perspectives, other than sequential reads, where BDMSs can take the advantage over SSDs, such as transaction processing as in (Lee et al., 2008) or exploiting the parallelism of SSDs as in (Fan et al., 2014). Li et al. (2010) argues that, a problem such as random writes are being even slower, can be optimised by other utilisation of an SSD-aware index structure.

The good news is that migration from rotational disk into SSD is considered easy to some extend. For example, there will be no modifications to be done on the level of code base. A conversion[1] through email has been made with Andy Seaborne from HP labs and the creator of Jena RDF store who suggested that B-tree index would not be affected by such a migration Seaborne (2014).

Indices are important data structures that BDMSs are already using for the purpose of optimising data access by order of magnitudes (Elmasri and Navathe, 2000). The general idea of indices is that data has to be structured in an optimal manner that provides fast access to data. Using indices will eventually increase the size of data being stored, as indices can be seen as another layer/copy of data. However, since modern HDs are spacious, a trade-off is, to some extent, acceptable in terms of space usage. The next sub-sections discuss known index structures in more detail.

Indices exist for the same purpose as an index for a book; instead of scanning the whole book searching for a specific chapter, which represents a linear scan $O(n)$, one can use the index, which should confirm whether or not that chapter exists. If it exists, it can provide the chapter page number. This index maybe in a hierarchical form/tree which can reduce the time to a logarithmic time *log(n)*, for example. For the purpose of this thesis, we will classify indices into two main general types: Ordered index and Hash index. Depending on the indexing technique, different factors need to be considered,

[1] `http://mail-archives.apache.org/mod_mbox/jena-users/201404.mbox/raw/%3C5355169D.4030102%40apache.org%3E/`

which also involves a trade-off. They are as follows:
Access Type: (RAM or Magnetic storage, etc): The architecture of the storage device hugely affects the way data should be indexed. For example, as RAM can point to an arbitrary page in a constant time, structures, such as hashing are more efficient. However, storages, such as HDs may benefit from the large space available, although the price of I/O is relatively high.
Time Complexity: Different operations can take different times to be processed, which can be measured by asymptotic notations. Usually, the big O notation is used as the upper boundary measuring the worst case, whereas the Omega notation (ω) is used as the lower boundary to measure the minimum time. Theta notation (Θ), which describes the tight boundary is a combination of the other two notations. For example, a Quick Sort Algorithm has an average and best case complexity of $\omega(n \log n)$, whereas its worst case can be denoted as $O(n^2)$.
Space Usage: The size of the potential index to be built also plays a huge role; current RAM are in the order of small gigabytes, so keeping data on the RAM is a decision that has to be taken carefully. The decision about deciding where the index has to take place may result in changing the whole structure of the index.
Index Purpose: (equality, range, spatial searching, etc): For example, a hash index maybe preferable for equality search, whereas a range search may be more efficient when using an ordered index.

2.1.2.1 Hash Index

Hashing refers to the operation of linking the original value with its hashed value. A key operation of hashing is what is called the "hash function", which determines the algorithm of transferring a key to an equivalent hashed value. Building hash indices usually comes in a hash map or key-value structure where keys represent the searchable entity, and the value consists of a hashed value or a bucket of values (such as, an array). An optimal case for hash indices can be seen within the RAM architecture. As RAM contains pages, each of which has its own pointer, that can be accessed in a constant time $O(1)$, a hash index can be built to perform a look-up also in also a constant time to access any key within that hash. Mainly, when building that index, each value of a key should be transferred into a hash through the hash function, which assigns the pointer to an equivalent original value. Clearly, the process of hash function can add a computability overhead to the overall performance when retrieving/generating a hash. Therefore, a fast computable hashing algorithm is usually required.

As data increases in size, it is also possible to store a hash table on multiple nodes. Inspired by the P2P systems, the Distributed Hash Table (DHT) was implicitly introduced in different systems, such as Tapestry, Pastry and Chord and then named by Ratnasamy et al. (2002). When the hash table/map exceeds the size of the main-memory, it is also

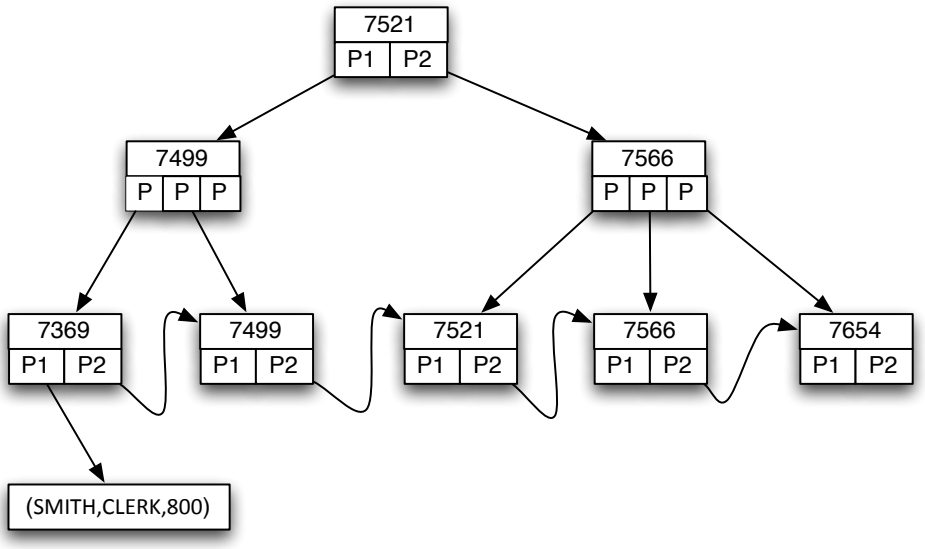


Figure 2.2: B+tree example for the data from Table 2.1.

possible to store it on disk. This can be done by dividing the hash index into segments, called Buckets, in a contiguous manner. When searching for a record, the entire bucket is read into memory (Samet, 1990).

2.1.2.2 Ordered Index

As the word "ordered" indicates, it is intended for such an index to be in a sorted order by some means. In a file that has been sequentially ordered, there can be either a primary index, which does not change the order of the file, also called a "Clustered Index", or a secondary index, whereby the search key specifies a different order from the original file order, also called "Non-clustered Index".

B-Tree is a self-balanced tree where each node can have multiple children. Each node can contain keys and values, whereas `B+Tree`, which is a special case of `B-Tree`, can only contain values at the leaves, which are the lowest nodes of the tree. `B+Tree` can also travel sequentially to other leaves. The major benefit of `B-Tree` is that when seeking a value, the time will depend on the depth of the tree, which usually is not comparable to reading from HD blocks when seeking random access. For example, the Figure 2.2 has a depth of 2. To retrieve data about an employee with an ID=7521, the pointer will travel only twice, rather than scanning all of the values, which would be the worst case scenario. Because of the tree nature, searching within a tree can improve the search time to a logarithmic time.

EMPNO	ENAME	JOB	SALARY
7369	SMITH	CLERK	800
7499	ALLEN	SALESMAN	1600
7521	WARD	MANAGER	2975
7566	JONES	SALESMAN	1250
7654	MARTIN	ANALYST	3000

Table 2.1: Employment data table

2.1.3 Query Processing

After data has been stored and indexed, the ultimate goal is to be able to perform querying tasks. Those tasks are usually divided into four main types: select, insert, update and delete. There are a number of general processes that are usually undertaken during the query stage within a SQL query (Garcia-Molina et al., 2009), for instance:

- Parsing: Constructing a parsing tree.
- Query rewriting: Converting the tree into an internal form (usually an algebraic representation). Then there will be an evaluation of a number of (logical) query plans to flag the cheapest one. For example, join re-ordering (such as a join with a higher selectivity may be executed first).
- Logical Evaluation: Here there will be an evaluation of a number of generated plans to choose the least costly one. Usually, this involves statistics about the data.
- Physical Evaluation: Here, also, there will be an evaluation of the number of physical plans for how to retrieve data from where it is stored, to also choose the least expensive one. For example, a hash algorithm may be used in preference instead of a nested for loop.

2.1.3.1 Relational Algebra

In the relational model, as data can be represented in a tabular form, an algebraic structure can be reasoned, allowing manipulation of data independently from its physical representation. Therefore, one essential step of processing a SQL query is to transform the query into a relational algebra or an equivalent expression. The ultimate goal of this step is that, for a given query, it is possible to generate different algebraic expressions, from which it is possible to choose the least expensive operation. Relational algebra is a procedural query language in which its steps have to be explicitly stated and followed (Silberschatz et al., 2010). This distinguishes it from the Relational Calculus, which is a non-procedural language and only describes the desired outcome without specifying the internal steps. The importance of relational algebra is its limited expressivity, which

makes queries simple to be automated and easy for a compiler to produce a highly optimised code (Garcia-Molina et al., 2009).

Using a relational algebra expression also leads to constructing an expression tree. Essentially, one expression may generate different trees, each of which representing a logical plan, making the evaluation stage easy to explore all the possibilities in a procedural manner.

2.1.4 Other Performance Considerations

When dealing with a BDMS, there are a couple of factors that should be carefully considered. In this section, we discuss some of these aspects where relevant to our study.

2.1.4.1 Caching: Mitigating Latency

When dealing with disk-based BDMSs, there is usually what is referred to as a "Buffer", where a dedicated in-memory space is allocated to the BDMS for its various operations. A BDMS will retrieve the data from disk, place them into the buffer, and then do other operations (such as intermediate joins or arithmetic) before sending the result to the user. The process is not completely repeated each time there is a BDMS query data; the buffer is usually cached by either BDMS or on an operating system level (DeBrabant et al., 2013). The concept of caching we are discussing here is the idea of lifting up data from one storage layer to an upper one. Specifically, keeping some data into the main memory for potential re-usage, instead of retrieving it again from the disk.

The challenge would then be: which parts of the data should be cached within a BDMS, keeping in mind the limited size of the buffer/cache. Different strategies have been proposed; a major use is to cache the upper level nodes of the `B+Tree`, which saves a considerable amount of time in disk accesses (Graefe, 2011). Frigo et al. (2012) introduced "Cache-Oblivious" algorithms; algorithms that do not know in advance about the size or number of levels about a given cache architecture. Other proposals cache result sets, such as Memcached (Fitzpatrick, 2004), a key-value in-memory distributed caching system.

2.1.4.2 Concurrency: Make the Most of Hardware

In a real world scenario, a BDMS should expect more than one client to be sending queries at the same time. Today's machines are becoming multi-cores, multi-CPUs, clusters or even multiple nodes that are located in remote places. This means that for a BDMS, it is important to make the most of the hardware's powerful characteristics

to increase their performance. Fan et al. (2014) present a system that increases the performance of read access to SSDs by means of paralleling table scans. Concurrency, however, comes at a price as concurrency programming has its own complications; from keeping threads synchronised across different processes to dealing with updates, for example.

2.1.4.3 Benchmarking: One/No Size Fits All

The performance of BDMSs is affected by various complex and interlinked factors. Some of these factors include: size of data, query complexity, data structure and hardware architecture. The factors are usually interlinked, which makes their performance evaluation even more challenging. With the rise of different BDMS models, such as RDBMS, RDF, NoSQL and many others, a user should have a clear expectation of a particular BDMS and its performance, which has led to various generic and specific benchmarks being proposed. In computer systems in general, Transaction Processing Performance Council (TPC)[2] is an organisation for benchmarking databases through TPC-C. However, it has been argued it is no longer a case of "one size fits all" (Stonebraker and Çetintemel, 2005; Kämpgen and Harth, 2013). The argument is that usually an apple is being compared to an orange, which also may lead to biased results.

2.2 String Matching and Indexing

String (or substring) matching is a well-studied problem within computer science in general. The problem is fundamental because of its various applications within computer science. It can be used for searching through text files, DNA/protein sequences, speech analysis, data compressions, image patterns, viruses signatures' and much more. Another area of string matching which has drawn much attention lately is Approximate Matching, which is discussed in more detail in the survey (Dorneles et al., 2011). This applies where the exact value is not known or is irrelevant. Our work is more about exact matching, rather than approximate matching, which is relevant to the main problem we focus on, Regular Expressions.

Definition 1. (String Matching Problem)
Given a pattern P with a length of m over a finite alphabet Σ, find all occurrences of P within a text T, where both $P, T \in \Sigma^*$.

For example, given a pattern P=/abcaba/, find all occurrences of P within the text T="aababcabaabc" , or declare failure. In this case, the pattern has one successful match with a shift {3,8}, where 3 represents the index of start matching and 8 represents where

[2] `http://www.tpc.org`

Algorithm	Pre-processing time	Matching time
Naïve approach	$O(0)$	$\Theta((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$\Theta(n+m)$
Finite state automaton based	$\Theta(m\,\lvert\Sigma\rvert)$	$\Theta(n)$
KMP	$\Theta(m)$	$\Theta(n)$
BM	$\Theta(m+\lvert\Sigma\rvert)$	$\Omega(n/m)$

Table 2.2: String matching algorithms and their complexity time (Saxena et al., 2011)

the index ends. A known use case is where a user is trying to find a word $W = P \in \Sigma^*$ in a text file T.

2.2.1 String Matching Algorithms

The literature of string matching algorithms is relatively large. Since 1970, around 90 new algorithms have been proposed, more than half within the last decade (Faro and Lecroq, 2013). These approaches can be divided into two main directions. Firstly, matching is done on the fly (on-line), where there are no pre-assumptions about P or T. In this case, it can be seen as a pure computational problem; sometimes there will be additional computation on P, but not T such as constructing an automaton. The other approach is to have an assumption about T, in which the data can be structured in a way which makes it more efficient to be searched. These would involve a pre-computation cost of T in the hope of increasing searching time afterwards, such as building tries.

Table 2.2 shows an overall comparison of the major algorithms and their time complexity in both pre-processing and matching time. Within the first approach, an intuitive solution has been called a "naïve approach" or a brute-force search. It does not require any pre-computations of P, that is to say, it simply scans and compares each character of P against T. The complexity of such matching requires $\Theta((n-m+1)m)$ with a constant space usage. The Knuth, Morris and Pratt algorithm (KMP) (Knuth et al., 1977), in a nutshell, assumes that when a mismatch occurs, P itself can determine where the next match could happen. It requires a linear pre-processing of m and a linear matching of n; making the overall complexity $\Theta(n+m)$, for the first time. Boyer-Moore (BM) Algorithm (Saxena et al., 2011), however, scans from right to left of P; the idea is that if the end of P is not matched, there is no need to match the first of P. This makes the algorithm do large jumps within T. This also makes the algorithm suitable for natural language editors. Moreover, BM works well with longer P. However, it is not preferable for smaller alphabets, such as binary or DNA sequences as in KMP.

2.2.2 Regular Expressions

The concept of Regular expressions (Regex) is one of the oldest topics in theoretical computer science. Stephen Kleene is considered to be the first who defined Regex (or regular set) by using an algebraic model (Kleene, 1951). Then, Regular expressions stayed in the context of mathematics until 1968, when Ken Thompson published the paper (Thompson, 1968) and its algorithm, later implemented within a text editor called QED. This text editor demonstrates whether a regular expression matches an input string or not. As Thompson designed and implemented the first Unix operating system, he also brought Regex into Unix, which in turn, was later adopted into various tools, such as Grep, Awk, Sed and others.

A Regular Expression (Regex) is a pattern that can be used to search for the existence of a sequence of characters. Regex is composed of two main parts: normal characters, or literals, and meta-characters which have special meanings. The importance of Regular expressions is when the exact value is unknown or irrelevant. For example, searching for all values which start-with "ali" can be represented in Regex as "^ali". This notion requires Regex to be highly expressive, unlike other simple string matching approaches.

Definition 2. (Regular language) A language L is a regular if and only if it is accepted by a finite state automaton (FSA).

Definition 3. (Deterministic Finite Automata) A Deterministic Finite Automaton (DFA) is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where

- Q is a finite set of states;
- Σ is a finite set of alphabet;
- $\delta : Q \times \Sigma \to Q$ is the transition function;
- $q_0 \in Q$ is the start state; and
- $F \subseteq Q$ is the set of accepted states.

A Regex pattern can be transformed into a DFA representation. Figure 2.3 illustrates how to represent a DFA for the regular expression /abc*f/

Definition 4. (Non-Deterministic Finite Automata) A Non-Deterministic Finite Automaton (NFA) is also a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where

- Q is a finite set of states;
- Σ is a finite set of alphabet;
- $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the transition function;

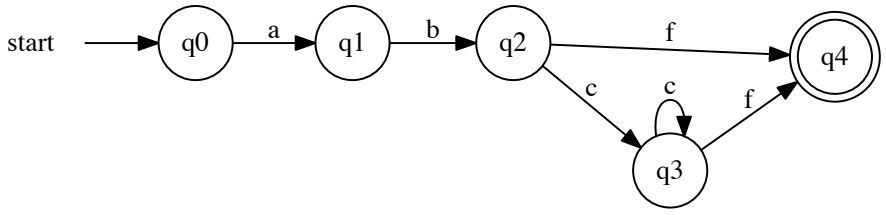


Figure 2.3: DFA for the regular expression "abc*f".

- $q_0 \in Q$ is the start state; and
- $F \subseteq Q$ is the set of accepted states.

Note that the only difference between Definitions 3 and 4 is within the transition function. In NFA, transition functions can be either elements of Σ or an empty string $\epsilon : \Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. Therefore, $\mathcal{P}(Q)$ represents the collection of all subsets of Q, that is the power set of Q.

Figure 2.4 shows NFA, another representation for the same regular expression "abc*f"

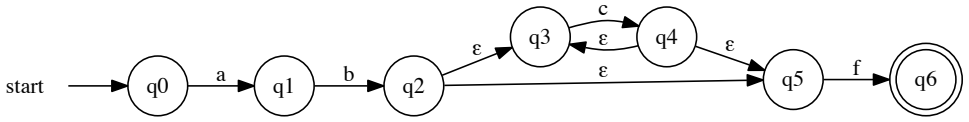


Figure 2.4: NFA for the regular expression "abc*f".

Definition 5. (Regular expression)

In regular languages $|L|$, a regular expression R can be defined as: R is a regular expression over a finite alphabet Σ where:

- **Alphabet Symbols:** where $c \in \Sigma$ is also a regular expression which have the length $|c| = 1$
- **Empty String ϵ :** where $L(\epsilon) = \{\epsilon\}$
- **Empty Language:** denoted by ϕ, and $L(\phi) = \phi$
- **Concatenation:** if A and $B \in R$, then $A.B$ is also a regular expression, and $A.B$ $\in R$, where $L(A.B) = \{vw \mid v \in L(A) \wedge w \in L(B)\}$

	Processing each character	Memory Space
NFA	$O(n^2)$	$O(n)$
DFA	$O(1)$	$O(\Sigma^n)$

Table 2.3: NFA versus NDA of matching each character and memory usage (Yu et al., 2006)

- **Union:** if A and $B \in R$, then $A \cup B$ is also a regular expression, and $A.B \in R$, where $L(A \cup B) = L(A) \cup L(B)$
- **Star Closure:** for a regular expression A, A^* is the star closure, and $L(A^*) = L(\epsilon) \cup L(A) \cup L(A.A) \cup L(A.A.A) \cup ...$

2.2.2.1 Regular Expressions Complexity

The idea is that for a given regular expression R, the engine will construct either a DFA or an NFA in $O(m)$ time. Once R has been constructed, the regular expression engine can match the literal string of length $|n|$ in $O(n)$ time against the FSA every time. Time results in a total complexity of $O(n+m)$ time. This is relatively efficient, given the high expressivity that Regular expressions provide (Gruber and Holzer, 2014).

Table 2.3 lists the differences between DFA and NFA in terms of processing each character of the matching string and the automata usage of memory. Although DFA and NFA are equivalent in power, which means that they all have the same expressivity in terms of computability, in practical applications, the choices between them have performance and resource differences (Becchi and Crowley, 2008). In a nutshell, DFA requires more computation while constructing the automaton $O(m)$ due to the high computation an engine requires, as DFA generates more states than NFA, which means that DFA requires more memory space. NFA, on the other hand, is faster in constructing time, as there are fewer states than an equivalent DFA. Nevertheless, a DFA ensures a linear time $O(n)$ in matching against a literal string. A DFA traverses states one by one at a time, meaning that the DFA cannot visit a state more than once, which causes matching to be $O(1)$ for each character within the literal string. On the other hand, NFA has a more complex structure, where transition functions can not only represent literal strings but also empty strings, making the regular expression engine do backtracking in some cases and, therefore, requires up to $O(n^2)$ when all states are active at the same time.

2.2.2.2 Regular Expressions Optimisations

Increasing the efficiency of Regular expressions has been around since Thompson introduced the Regex algorithm. There are two main approaches to Regular Expressions optimisation discussed in the literature. Mainly, researchers are looking for a trade-off

between space and time efficiency from a computational perspective on the fly. The other approach requires pre-processing of the data, that is to say, through using special data structures (indices) which can increase the efficiency of matching strings.

The first approach has been investigated more in the context of Deep Packet Inspections (DPI), as network filtering involves high computations on the fly. The motivation for this part is taken from the argument discussed in Section 2.2.2.1. Using DFAs, Kumar et al. (2006); Becchi and Crowley (2007) proposed heuristics to compress DFA transition tables. Yu et al. (2006) introduced combining Regular Expressions within different DFAs, or extending finite automata with variables as proposed by Smith et al. (2008). On the NFA side, Yang et al. (2011) implemented Ordered Binary Decision Diagrams (OBDDs) to process a set of NFA states.

Building indices, however, has been mostly studied in the context of searching string patterns within large corpora. The main idea is to pre-process the data beforehand; usually, both the original data to be searched and the new index are disk-resident. As Regex requires both Regex patterns and literal strings to be in-memory for the engine to start matching, within large data, both disk access and the matching process will take a long time. Therefore, building special indices, which the engine can first consult, gives a major improvement on the overall time. A drawback of this approach is the expected index size, as Regex is highly expressive, making building such indices even more challenging. Cho and Rajagopalan (2002) proposed a multi-gram index to reduce the set of searchable strings (named FREE). As the index is expected to be very large, they used a selective algorithm for grams based on their frequency. RE-Tree (Chan et al., 2003), however, takes an opposite approach, where for a given literal string s, it finds all Regular expressions $|R|$, that satisfy s. It uses a highly balanced tree which stores bounded automata in its nodes. Chen (2012) extended FREE for less index building time, added a distance based gram and advancing the querying capabilities through deeper analysis of Regex patterns.

2.2.2.3 Modern Regular Expressions

Regular expressions are widely used for many purposes other than searching through text. They can be used for validating users' entries, such as email or usernames. They are also used within anti viruses for detecting virus signatures. In networks, they are extensively used within filtering tools or Deep Packet Inspection (DPI) (Yu et al., 2006), which inspect viruses, spam or controlling traffic. They are also used for matching DNA and protein sequences as well. Above all this, Regular expressions play a huge role in designing programming language compilers.

This demand causes programming languages to adapt them as tools. However, programming languages, such as Perl, have taken Regular expressions into new levels of

EmployeeID	Name
2344324	Abbey Smith
3455346	Saud Aljaloud
4345345	John Harris
2342344	Smith Williams
2323423	Mike Adams

Table 2.4: Employment table showing their IDs and names

features/flavours that are not even considered within regular languages. The issue regarding these extensions is that it becomes difficult to study them theoretically. A known feature is back referencing, where a sub pattern is matched more than once through the same regular expression. The pattern, for example "([0-9]+)\:\1" will match a sequence of numbers repeated twice with a colon between them such as "23:23". Another issue with this kind of Regular expression is that these are not included within regular languages nor with context-free languages. These Regular expressions are commonly referred to as extended Regular expressions.

2.2.3 String Matching within DBMSs

In this section, we discuss the problem of matching strings, including Regular expressions, where data is structured and stored on an external commodity such as hard disks.

Definition 6. (String Matching within DBMSs problem)
Given a string pattern P that is submitted as part of a structured query Q to a persistent BDMS, find all records R that satisfy both P and Q.

Given Table 2.4, find all names that have the pattern "Smith". There are two main problems associated with such a query. Firstly, all names have to be fetched from hard disk and placed in the main memory; meaning there will be a full scan of records. In the case of numerous records, this causes expensive disk accesses. Secondly, there will be an additional computation involved for the string pattern matching. To make it even slower, patterns can be expressed in the form of Regular expressions.

2.2.4 Inverted Index

One of the main methods in pre-process literal strings is to apply an inverted index over the text. Inverted indices are also a broad topic, involving involves different data structures and implementations. However, they mostly will have a `dictionary` and a `postings list(s)` as the main components. For example, given a collection of documents D, each of which has a set of words W, an inverted index will list all the unique words in its dictionary. Each of the words in the list will have an associated postings list, which contains a pointer to the document which contains the word. This

is where the word "inversion" comes, as users are looking for documents D, they are internally listed by words D. This can provide a powerful keyword search, for example, where a user can insert a set of keywords, and then the application can return the set of documents which contain these keywords. Keyword searching or Full Text Searching (FTS) has been a centric technique for many applications. It is simple for users to write and can be extended to provide a much more powerful tool. For example, it is possible to add the position of the word within the document to the postings lists to provide an approximate search. A dictionary list can be normalised with other related words using techniques such as stemming.

Document ID	Content
1	Inverted index is fast
2	Fast string search
3	Inverted index for FTS

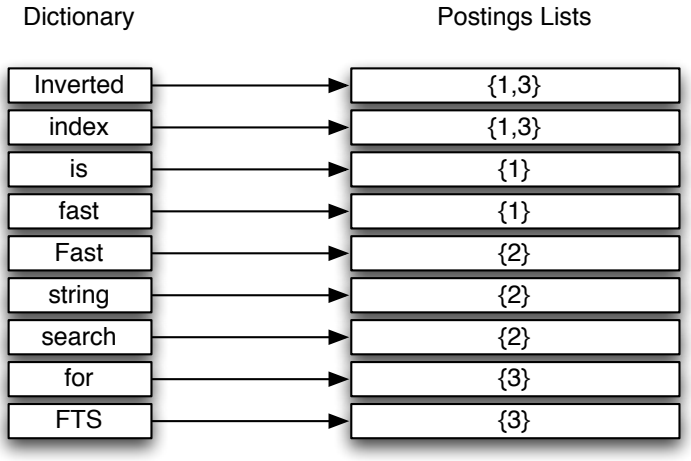


Figure 2.5: Inverted index architecture.

2.2.4.1 Inverted Index within DBMSs

Searching string patterns or Regular expressions over a BDMS is not a trivial task, especially in the era of big data, which does not all fit into memory. Nevertheless, it is rare for a BDMS not to have a FTS capability, for example. This area is one main example where both BDMS and Information Retrieval (IR) get together to solve the same problem. The main challenge is usually the mismatch between the structural language and data storage used by the BDMS and the inverted index. A popular way of implementing this technique is by integrating a specialised FTS engine within the BDMS, such as (Amer-Yahia et al., 2004) for XQuery, (Hamilton and Nayak, 2001; Agrawal et al., 2002) for SQL or (Minack et al., 2008) for SPARQL.

Others make use of the inverted index for string patterns and Regular expressions. Stuhr and Veres (2013) built an inverted index for speeding up Regular expressions within

SPARQL. Lee et al. (2011), however, employed a `B-Tree` inverted index for reducing the result set to be matched with the Regex engine.

2.3 The Semantic Web

The Semantic Web (SW) is a broad subject that is meant to apply a paradigm shift from a web of documents to a web of data, started in 2001 by Berners-Lee et al. (2001). The intention is that machines should be able to process, understand and share data on the web with each other, using collections of Semantic Web technologies, that is to say, by using ontologies as the base for deriving knowledge (Hendler, 2001). The web, as it has been, suffers from a huge amount of data that is not easily interlinked and, therefore, it cannot exploit the relationships between data; integration between systems is known to be a costly job and causes different layers of middleware services that only exist for this purpose, and different services may need their own models as well. The long term goal of the Semantic Web is to allow agents to be able to openly communicate with each other with minimal human intervention. In the health care domain, for instance, it might be easily possible to analyse the relationship between a disease within an area in relation to the weather status during the disease appearance with other parameters using a single Semantic Web query with an appropriate visualisation.

Nevertheless, there are a number of issues which have to be carefully addressed. The web has become not only a field that can be dealt with by using traditional computer science paradigms, such as engineering or mathematics, but it is now more about a broad interdisciplinary field which brings together these areas with others, such as sociology, law and psychology (Shadbolt et al., 2013).

A major issue of the web, in general, and also the Semantic Web, is the web of heterogeneity. It is difficult, sometimes judgemental, to conclude that two statements are referring to the same thing. Technically, this happens when two statements (triples) are describing the same thing with different identifiers (URIs). Notably, it cannot be guaranteed that people/agents will use only one URI for a specific thing; in fact, it is especially likely for people to instantiate their own statements for various reasons: they may not know that such a resource exists, or they may want to express their own vocabularies/opinion in their way. A major approach to solving this problem is to use a co-reference mechanism, named "owl:sameAs" property, which allows the linking of two resources as being identical. To some extent, a machine may be able to distinguish some similar resources using techniques from Information Retrieval. However, for example, two words may be similar but have different meanings, which brings us back to the big question that the Semantic Web has to answer. Another major issue is with how much certainty one should treat such statements. In the previous example, assume that two different resources have been linked by someone/agent as being sameAs. Further

argument about the usage of owl:sameAs by Halpin et al. (2010) proposed a provenance model to be added as a fourth dimension to the usual triple, which represents where the triple came from.

However, businesses play a huge role in technologies emerging and being developed. For example, as stated previously about the relational model, it was the banking domain which introduced that model; Oracle, Microsoft and IBM are all profit organisations who have introduced state-of-the-art BDMSs. Although some specific-business models have been supporting the Semantic Web, it is still questionable as to why there is no general and clear business model for enterprises. Consequently, for example, Google introduced their own closed model, which they call *the Knowledge Graph*[3], which seems to have some of the Semantic Web visions, yet with very little acknowledgement of using any of the Semantic Web technologies. The same situation applied with Facebook, when they introduced their *Graph Search*[4]. This may be because of the principle of the Semantic Web being open, which may not be preferable to businesses.

2.3.1 RDF Data Model

The Semantic Web is hugely based on the expressive Resource Description Framework (RDF), which was first proposed in 1999 (Klyne et al., 2004), where statements should be represented in the form of triples (Subject - Predicate - Object). A collection of RDF statements results in a directed and labelled graph, which makes statements extensible, interoperable and supports decentralisation following the general W3C principles[5]. For example, a statement like "Saud Al-jaloud has a first name, which is Saud" can be expressed in an RDF graph as in Figure 2.6.

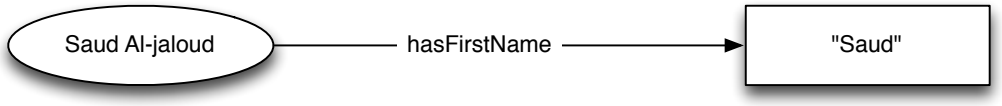


Figure 2.6: Single RDF triple graph

However, the Semantic Web, in its whole and initial vision, was criticised for not being completely practical. This led The World Wide Web Consortium (W3C[6]) to set four main principles (Berners-Lee, 2006), which are:

- Name things with URIs
- URIs should be in HTTP format, so they can be referenceable/dereferenceable

[3] http://www.google.com/insidesearch/features/search/knowledge.html
[4] https://www.facebook.com/about/graphsearch
[5] http://www.w3.org/Consortium/mission.html
[6] http://www.w3.org

- Add useful information about the URIs using standards like RDF/SPARQL
- When writing RDF, refer to other URIs as much as possible

From Figure 2.6, and by using these principles, it is possible to extend another detailed graph as in Figure 2.7.

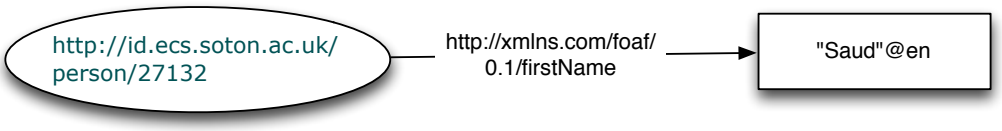


Figure 2.7: More detailed RDF graph of Figure 2.6

For more elaboration on the two graphs, both resources and the edge have been updated. In the first graph, Figure 2.6, we used a blank-node to represent Saud_Aljaloud. This mainly helps in simplifying manipulations of resources locally, rather than having to link each resource or having to globally identify them, which may seem exhaustive. A better usage, however, as suggested by the Linked Data principles, is stated in the second graph. In Figure 2.7, we used a URI, in an HTTP format, from another source to represent the resource Saud Al-jaloud.

Similarly, the predicate hasFirstName from the first graph, Figure 2.6, which was locally produced, a better representation would be produced by using a standard vocabulary. In the second graph, Figure 2.7, we used a well-known vocabulary called FOAF[7] to represent the FirstName property. For simplicity, prefixes also can be used to reduce the length of the URI. For example, in the graph, Figure 2.7, it is possible to express the predicate as `foaf:firstName` by referring to the URI of the prefix at some point as `http://xmlns.com/foaf/0.1/`, in this case.

Literal nodes, such as "Saud" and "Saud@en" or numbers, to some extent, are not considered as a resource node, as they are self-explanatory. That explains representing them with rectangles instead of circles. They also cannot be expanded by being the subject of other nodes, placing them at the end of the graph leaves. In other words, Literals cannot be subjects: they are always objects. In the second graph, Figure 2.7, we just added a language tag for more details about the value. It also can be conceptualised that literal nodes introduce the unstructured bit of data into the Semantic Web.

These principles have to be complied with to ensure a high level of benefits under the name "Linked Data", which, as Berners-Lee said "*Linked Data is the Semantic Web done right*" [8]. One of the main principles, and unlike other data models, is using Uniform Resource Identifiers (URIs) to reference/dereference things on the web. Since

[7] `http://www.foaf-project.org`
[8] `http://www.w3.org/2008/Talks/0617-lod-tbl/#(1)`

URIs ideally have to be unique, pointing to things with a URI will mainly reduce the heterogeneity that the current web has when things are referred to/at from other sources. In other words, URIs for the web of data can be seen as just URLs for the web of documents, thus making the granularity of processing data on the web more finer and more flexible. This results in a community-effort, the so-called LOD Cloud, which contains 295 datasets, each of which has to be linked from/to at least another dataset, as of the last update from 09/2011[9].

In the database community, the RDF model has been argued in relation to graph theory in (Angles and Gutiérrez, 2008), the Relational Model (Faye et al., 2012), and distributed systems (Stuckenschmidt et al., 2004).

2.3.2 RDF stores

Triples in their basic form of (S - P - O) have to be properly stored to ensure effective querying and manipulation results. The main assumption about triples schema is that they are expressed in three parts/columns, which increases the unpredictability of this model, in relation to relational model, for example. This unpredictability bears overhead in designing the generic RDF store. Moreover, the volume, velocity and veracity of data on the web is increasingly challenging the performance of RDF stores. One of the goals for the Semantic Web Challenge[10] was dealing with a large volume of data. In 2013, one of the two tracks for the challenge was called the "Big Data Track", and one in 2012 was called "Billion Triples Track". A trivial approach is to rely on mature systems, mainly RDBMS-based systems. However, there is a clear mismatch between the two models (RDF vs. Relational Model). Therefore, it is possible to classify these proposals into two main categories: non-native and native triple stores (Faye et al., 2012).

2.3.2.1 Non-native Stores

Since SW repositories are relatively new, where most of the databases are in a prototype research state, such as RDF-3X, the obvious direction would be to use mature systems. Relational databases (RDBs) have been an interesting model since the emergence of the Semantic Web, for example, MySql, as in 3Store (Harris and Gibbins, 2003), PostgreSQL, as in (Wang et al., 2010) or DB2, as in Jena SDB (Wilkinson et al., 2003). This is mainly because of the assumption that the RDF model can be completely manipulated as a logical model without the need to propose a new physical one. At a higher level, RDF data can be seen as a *relation* that includes three *attributes* representing: *Subject*, *Predicate*, and *Object*, respectively. However, this is not a straightforward transformation. Indeed, there has to be a mapping mechanism between the two models. W3C has set

[9] http://lod-cloud.net/state/
[10] http://challenge.semanticweb.org

up a dedicated group named "*RDB2RDF Working Group*"[11], which aims to standardise languages for mapping between the models. Two languages were recommended: Direct Mapping (DM) (Arenas et al., 2012) and RDB2RDF Mapping Language (R2RML) (Das et al., 2012). Hert et al. (2011) proposed a comparison between the previous two with others in a feature-based comparison. Their compulsion is that the choice between the mapping tools should depend on the purpose of the mapping and, therefore, they classify mapping tools into different types.

Furthermore, when using a RDB, normalisation can be used, mainly in cases such as *predicates*, to be stored in another *relation*. A similar model is proposed by Harris and Gibbins (2003). A major barrier would be the drawback in performance; mainly that a SPARQL query will have to be translated into a SQL query. Moreover, the nature of the SPARQL query, from a SQL perspective, is that the query will produce many joins (Sahoo et al., 2009). Another model, a property-oriented model, was proposed by Abadi et al. (2009). A major issue of the later model is the difficulty of dealing with Null values, where not all resources will have the same predicates.

3Store 3Store is one of the early RDF stores that was proposed by Harris and Gibbins (2003). It was built using C Library and can be classified as a non-native store, since it runs on top of Mysql database with the usage of the normal form of single table, consisting of (Model, Subject, Predicate, Object), where Model denotes the origin of the triple (where triples come from). Moreover, all values in the table are being hashed and referenced by other data tables that have resources and literals to reduce the storage requirements on the main single table. Then, they developed a query translator from RDQL to SQL, as SPARQL did not exist at that time. Having done this, evaluation of queries is done in the underlying database, rather than the RDF part. Therefore, although it is possible to benefit from a mature RDBMS, in terms of the sophisticated query optimisations by Mysql, while not all RDF query languages have their equivalence completely in a SQL format.

Virtuoso is one of the main BDMSs that have been used by large repositories, such as the LOD2 project (Boncz et al., 2014). It also shows a high performance across different benchmarks, such as the BSBM benchmark.

Virtuoso is built on top of its original RDBMS architecture, with various optimisations with regard to the RDF model. It comprises a table of four columns, which represents the three triples, with their graph as the fourth dimension. Other tables are used to map different IRIs and literals with ID values. Indexing permutations are less used with Virtuoso, by relying on two main orders: GSPO and OGPS. The latter allows the usage of a bitmap index, where S is represented by IRI ID, unlike O which has a much larger

[11] `http://www.w3.org/2001/sw/rdb2rdf/`

cardinality (Erling, 2006). For querying, SPARQL is translated into SQL during the parsing phase. For query evaluation, statistics are hard to apply, since data is stored on a main quad table, rather, it relies on analysing samples during the translation phase to get the best evaluation plan (Nitta and Savnik, 2014).

2.3.2.2 Native Stores

As the logical models of RDF and the relational model are fundamentally different, some triple stores applied their own physical models directly without using the relational layer in between. These stores try to benefit from techniques which RDBMS are based on. Mainly, `B+Tree` is used as the base for storing RDF triples in its six query patterns (*SPO, SOP, OSP, OPS, PSO, POS*); each pattern has its own index, as used by YARS2 (Harth et al., 2007) or later by RDF-3X (Neumann and Weikum, 2008). As these stores are new, they most probably miss some optimisation techniques or properties such as recovery, ACID properties and so on. However, in terms of the basic querying of RDF, they, in general, prove to be faster than the non-native stores. Unlike non-native stores, which usually include different mappings from RDF to RM, such as from SPARQL to SQL, native stores use a direct mechanism for such a specific requirement.

Jena (Carroll et al., 2004) is a Java-based framework that has been designed to enable the manipulation of Semantic Web data on different levels, such as storage, applications and integration with other tools. Unlike most of the other stores, Jena provides a Java Programming Interface for the RDF model. This helps applications such as Protege to be built on top of Jena to provide easy building of ontologies and visualisations of data.

One of the services that Jena provides is the ability to store RDF data through different mechanisms (Wilkinson et al., 2003). In addition to the in-memory storage, it offers both native, by Jena TDB, and non-native, by SDB storages. The SDB storage uses different RDBMSs, such as Mysql, PostgreSQL, and many others[12], through their JDBC drivers. As with most other non-native storages, it stores each triple in a single row which contains the usual four columns (Subj, Prop, Obj, GraphId) with the use of two indexes for (subj, prop) and (obj).

Jena TDB[13], however, employs its own mechanisms for storing data and indices with non-transactional operations. This also led TDB to apply its own query optimisations using two strategies: fixed or statistics. The fixed strategy looks only to the variables within the SPARQL for its reordering decisions. The statistical approach is more like the evaluation plan in a conventional SQL, where statistics about the triple patterns are collected, either manually or by the engine to be calculated before reaching a decision.

[12] `https://jena.apache.org/documentation/sdb/databases_supported.html`
[13] `http://jena.apache.org/documentation/tdb/`

RDF-3X is a native and open source RDF engine that was proposed by Neumann and Weikum (2008). RDF-3X is completely dependent on the `B+Tree` structure as the origin of data, rather than additional index lookups. Specifically, it generates six compressed and clustered indices to represent the six different forms of triples (such as : SPO, SOP, POS, etc). For example, consider the query [(Smith, freiendOf, ?X)]: to answer such a query, RDF-3X will refer to the index of order SPO, which has SP as a key. In this case, the key is [(Smith, freiendOf)] and the value is O, thus, the result values are all at the leaves applied to the stated key. Moreover, there is a merge-only join for performing joins, rather than the well-known sort-merge joins algorithms, as a result of the pre-sorted `B+Tree`.

Sesame was first proposed by Broekstra et al. (2002), and was then developed by Aduna[14]. Although its name has been lately changed to (RDF4J), this thesis will use the mane (Sesame), as experiments have been applied on that version. In a sense, Sesame is similar to Jena for parsing, storage and manipulating Semantic Web technologies, such as RDF, OWL and SPARQL, using an open source framework. It is also known for its Storage and Inference Layer (SAIL) API, which allows different triple stores (back-ends) to be built underneath this API, such as OWLIM, BigData and others[15]. SAIL provides an easy solution for developers to integrate different APIs using this layer.

Owlim-SE is the commercial edition of OWLIM rdf store, developed by Ontotext and built using the Sesame SAIL API. A newer version of this store has also been renamed to (GraphDB). As with most triple stores, Owlim-SE also uses permutations of an indices strategy. FTS is performed using either Lucene-based implementation (aka RDF Search) or proprietary implementation (aka Node Search)[16].

4Store (Harris et al., 2009) was developed and is maintained by Garlik[17]. It was built using ANSI C99 and licensed with the GNU General Public version 3. 4Store is mainly designed to work on low-cost servers, using a "shared nothing" and prefers to have all the indices in-memory. It has been reported that 4Store is able to handle 15×10^9 triples on a production cluster[18]. Triples are indexed using quads, and both radix tries and hash tables are stored as indices to be used, whether in-memory or as random access is required, respectively (Nitta and Savnik, 2014).

Rasqal is used as a SPARQL parser, which also provides a parsing tree that is used by 4Store's relational algebra engine. The main query optimisation of 4Store is join ordering, but it also employs common subject and cardinality reduction optimisations.

[14] `http://www.openrdf.org`
[15] `http://www.openrdf.org/documentation.jsp`
[16] `http://owlim.ontotext.com/display/OWLIMv54/OWLIM-SE+Full-text+Search`
[17] `http://www.garlik.com`
[18] `http://www.w3.org/wiki/LargeTripleStores`

Bigdata (systap, 2013) is an open source RDF store that was built using the Sesame SAIL layer, licensed using the GPLv2, implemented by Java and maintained by Systap[19]. The name of this store has also been renamed to (Blazegraph), we keep calling it (Bigdata), as experiments was carried out using the previous Bigdata version. As with 4Store, Bigdata is designed to work on clusters, including a single node, using Apache ZooKeeper which facilitates the distribution coordination between nodes. This allows Bigdata to scale well horizontally. The main data structure for its index is the `B+Tree` with highly configurable factors, such as branching factors, page size and buffer mode. When used on a single machine, the index has a different structure from a cluster one. A single node index may not be able to run some "unselective queries" as quickly as when the data is on a cluster, as a cluster index stores data in a key-order manner, which allows a sequential scan rather than random access. Querying mainly relies on join order optimisations. This also means that "selective queries" are run faster on a single machine, as there will not be a coordination overhead.

Stardog[20] is a commercial RDF store, which has been relatively recently released and is reported to be able to handle 50×10^9 triples. It is written purely using Java. Stardog has not been as active within the research community as other discussed stores, which makes it harder to study the internal design, other than treating the store as a black box while experimenting.

2.3.2.3 Benchmarking RDF stores

Various benchmarks have been proposed, due to the high expressivity that RDF provides, which makes different RDF stores perform differently in various cases. Duan et al. (2011) argue that benchmarks such as BSBM (Bizer and Schultz, 2009) or LUBM (Guo et al., 2005) are structured in an RDBMS-style, which may give some RDF stores that are optimised to this structure a better performance. They also argue that because they are mainly based on synthetic data, in which they follow automated rules, they will still follow an RDBMS-style. The other approach is to have both real-world dataset and queries. Morsey et al. (2011) analysed the log files of DBpedia endpoint, which has all queries executed by real users. Then they clustered and reduced them to a set of queries to be executed against the DBpedia dataset.

2.3.3 SPARQL Query Language

The way data can be queried on the Semantic Web is a huge advantage compared to the traditional web (Berners-Lee et al., 2001). Prior to SPARQL, there was no standard

[19] `http://www.systap.com`
[20] `http://stardog.com`

language for querying RDF. Languages, such as RQL, SeRQL, TRIPLE, RDQL, N3 and Versa have been discussed in (Haase et al., 2004). Subsequently, the W3C published the recommended language for querying the Semantic Web named SPARQL, as proposed by Prud'Hommeaux and Seaborne (2008) as an updated version.

SPARQL 1.1 (Seaborne et al., 2013) is the latest version, and includes some missing important features, such as Property Paths, a major graph model property for traversing through nodes. Mostly, SPARQL 1.1 is more about standardising the syntax for different features that have not been strictly specified, such as how to load bulk RDF, and string-related patterns, such as STRSTARTS, STRENDS or CONTAINS and others.

It is also intended that SPARQL works openly. The W3C also published the SPARQL Protocol for RDF (Feigenbaum et al., 2013), which allows communications over the internet for sending/receiving SPARQL queries via what is called a SPARQL Endpoint in the form of XML, JSON, RDF/XML CSV and other formats. This, technically, allows the idea of a Federated Query, one of the new features in SPARQL 1.1, for querying different sources of RDF stores by sending one query across different datasets.

SPARQL works by specifying triple patterns, embedding variables as a Basic Graph Patterns (BGP) and filters to be matched from within the RDF store. The main benefit from using SPARQL is that both machine and human can send a query against one or more SPARQL end-point(s) and then retrieve a machine/human result.

SPARQL, as a query language, has many characteristics in common with SQL. Cyganiak (2005) discussed a transformation from SPARQL into both Relational Algebra and SQL. It also describes issues about the mismatch between the semantics in SPARQL and the general Relational model. This is based on the assumption that both SPARQL and SQL can be modelled as tuples.

2.3.3.1 Searching the Semantic Web

Searching through strings within the SW has been studied from different perspectives, as discussed by Wei et al. (2008); Jindal et al. (2014). Proposals such as SWSE (Hogan et al., 2011), Swoogle (Ding et al., 2004), Sindice (Tummarello et al., 2007), Watson (d'Aquin et al., 2008) and many others have been formed around the idea of building a search engine in which a user can search for resources that have a relation to a string, usually with keywords that involve natural language processing, such as stemming, and a ranking mechanism. This kind of approach is query-independent, where a user would be able to submit simple keywords. Systems usually provide a means of navigation for the user to find relevant resources.

The other umbrella of searching through SW is the SPARQL-specific approach. SPARQL, as the standard SW querying language, contains standardised functions to search through

strings such as Regex, CONTAINS, STRSTARTS and other XPath string functions. Others also adopt unsupported features, such as Full-Text Searching (FTS).

2.3.3.2 SPARQL Query Optimisation

Query optimisation, in general, is one of the main factors that can affect the efficiency of a system querying performance. Within SPARQL, this is an even more critical factor. Mainly, the nature of the data model (RDF), which has a high expressivity through its free schema, causes a major challenge. For example, SPARQL produces more joins than an equivalent SQL query, for example. There are several research studies that have been focusing on the fundamental properties of SPARQL, such as (Schmidt et al., 2010; Pérez et al., 2009). The former paper shows that the OPTIONAL operator is PSPACE-complete.

From one perspective, SPARQL has the notion of end-points; a system is supposed to reveal a SPARQL query facility to other developers or machines, who may not be concerned about whether the performance should be able to submit raw SPARQL queries. Other models, such as SQL, do not reveal direct access to the database engine. Rather, someone who has a full access, such as a developer or administrator is the one who writes the query, which at least controls the general structure form of a query. By doing this, a part of the optimisation can be done by these developers or administrators. For example, a developer may opt to use LIKE rather than the Regex function, as LIKE is less expensive in terms of computations, for simple search dialog that an end-user may use through a text box on their website. In this case, SPARQL is more complex because of the notion of end-points. Moreover, because of the high expressivity provided by the RDF model, querying can be also expensive. The price of these services/expressively, from one point, is not just that query optimisations have become even more critical, but also automating them, as end-points is used.

Cost-Based Optimisation:
Cost estimation is the process of choosing the least costly plan, according to the cost of the various operations that are going to take place within a query. Mostly, it aims to estimate the cost of consumed memory or disk and the processing time. For instance, some joins are preferred to be done before others. Join ordering is a major optimisation within SPARQL, since SPARQL is known to produce many joins. Heuristics are usually used (Tsialiamanis et al., 2012).

Histograms:
Histograms are mainly used for estimating the result set of an operation based on the distribution of data. Using histograms can help to choose the least costly ordering of

joins. There are different known types of histograms, such as equal-depth, equal-width and most-frequent-values histograms. Constructing these histograms may be expensive, but usually they are not done simultaneously within each update. Rather, they are done on a scheduled basis, or even manually. This is because the effects of histogram changes are not usually dramatic, but accumulative over a long period. An approach is based on selectivity estimation (Stocker et al., 2008).

```
SELECT *
WHERE {
    ?book dc:author ?author .
    ?book dc:price ?price .
        FILTER(?price > 30)
}
```

Listing 2.1: SPARQL query with a numeric XPath Test Filter.

Optimising FILTER Operators:
In general, it is known that pushing filters (such as logical filters) into sub-expressions is considered to be a good strategy. Filters can reduce the intermediate results before a join can be done. Consider the query from Listing 2.1, which shows a SPARQL query example that asks for books whose prices are greater than 30. Figure 2.8 shows two expression trees that can be constructed from this query; the one on the right performs the filter first on the relevant triple pattern. The figure on the left performs the join first. In this example, it is preferable that the intermediate results are reduced before the join is applied. Section 2.3.3.3 discusses the Regex filter operator in similar cases.

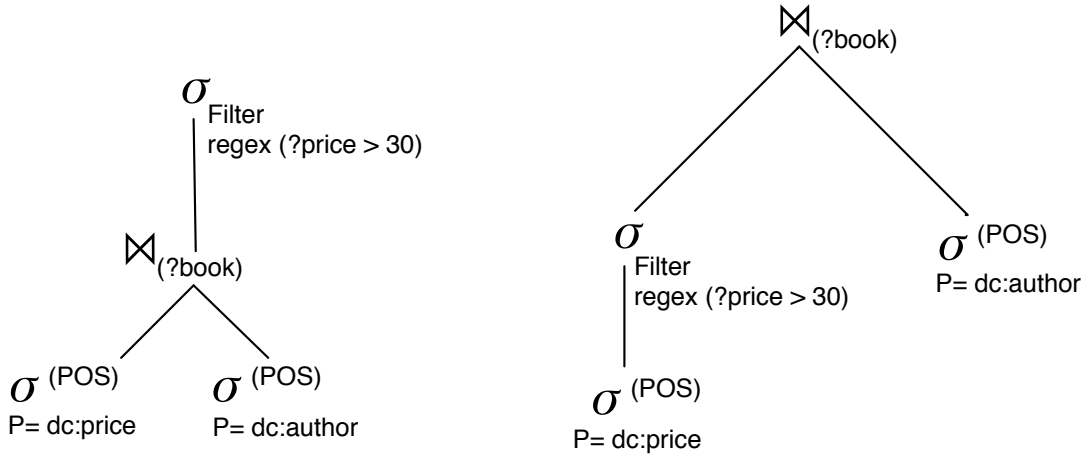


Figure 2.8: Expression trees of Pushing filters within SPARQL.

Other optimisations of filters are done by eliminating the filter. Filters, in general, are expensive, in that they require a full scanning of records. For example, a query similar

to the one above, with equality instead of the range filter, such as: FILTER(?price = 30), would be better rewritten as: ?book dc:author ?author . ?book dc:price 30. Figure 2.9 illustrates the expression tree of this query. If the variable ?price is being projected, it is also possible to add BIND (30 AS ?price).

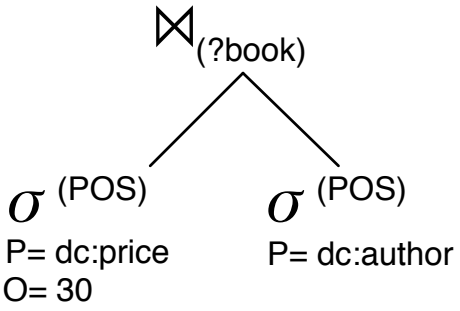


Figure 2.9: Expression tree for eliminating the filter expression.

2.3.3.3 Regex within SPARQL

Regular expressions are the main function to search through strings within SPARQL. Regex are applied within a filter expression alongside other filter operations. In terms of standardisation, Regular expressions are a good choice, because of their long history in the literature and their clear formal definition. They are already being used with similar standards, such as XML schema, XPath and XQuery. These standards are being already used within SPARQL. Moreover, Regular expressions are highly expressive languages for searching string patterns. They have been embedded within even the first editors on computers, such as Vi, Emacs and most other current editors.

This, however, comes at a price; a highly expressive language requires high computation levels. In terms of SPARQL and other BDMSs, it is difficult to reduce the intermediate results before records are fetched from the hard disk in an efficient way. Regex requires all records to be in the main memory. This causes two main problems: current rotational media have a high latency due to their seek time. As all records are being presented in the main memory, these records have all to be evaluated within the Regex engine as well. A major mitigation of these two challenging factors is to reduce the number of records to be processed. This reduces both the seek and evaluation times.

Logical Optimisations:
Optimising Regex is not as straightforward as optimising other filter expressions. Regex requires deeper analysis of the Regex pattern itself for logical optimisation. Here, we describe some different optimisations that can be done on Regex queries. Consider the query: ?product rdfs:label ?label . FILTER regex (?label , "`^keyword$`"). This query

can be optimised as: ?product rdfs:label “keyword” . BIND (“keyword” as ?label). Figure 2.10 shows both the native expression tree on the left, and the optimised one on the right. The figure on the left has the main requirement of scanning two variables (OS), unlike the figure on the right, which only requires scanning one variable (S). This has a major impact on the number of records to be scanned within the index (POS). Moreover, there will not be any further filter evaluation which is also an expensive operation, especially within a Regex engine. The Regex pattern, however, has to be only with these two meta characters. Other patterns which have more complex structure, such as “`^ke.*rd$`” should not be converted. In the same way, other Regex patterns, such as “`^keyword`” or “`keyword$`” can be rewritten as XPath functions with STRSTARTS or STRENDS respectively, as shown in Figure 2.11. We carried out an experiment regarding the performance of Regex and XPath, which is discussed in Chapter 4.

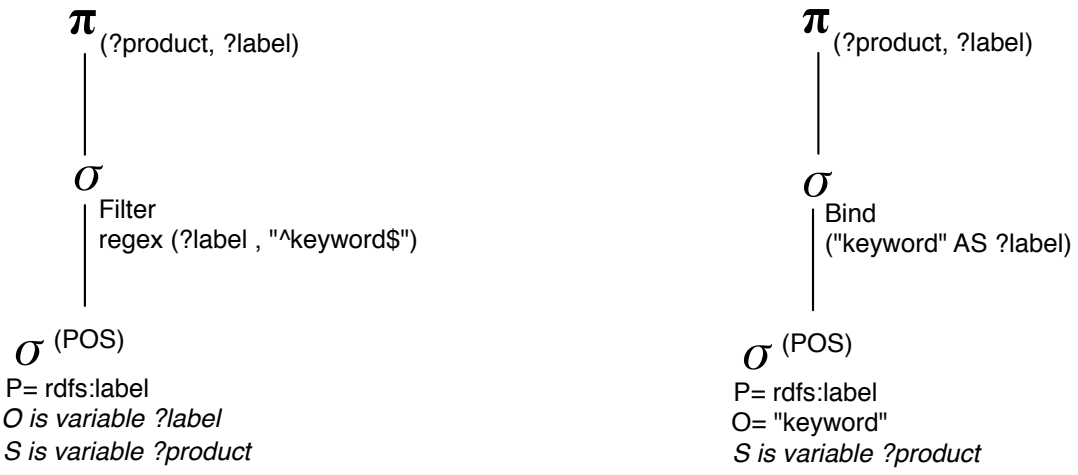


Figure 2.10: Expression tree for eliminating the Regex filter.

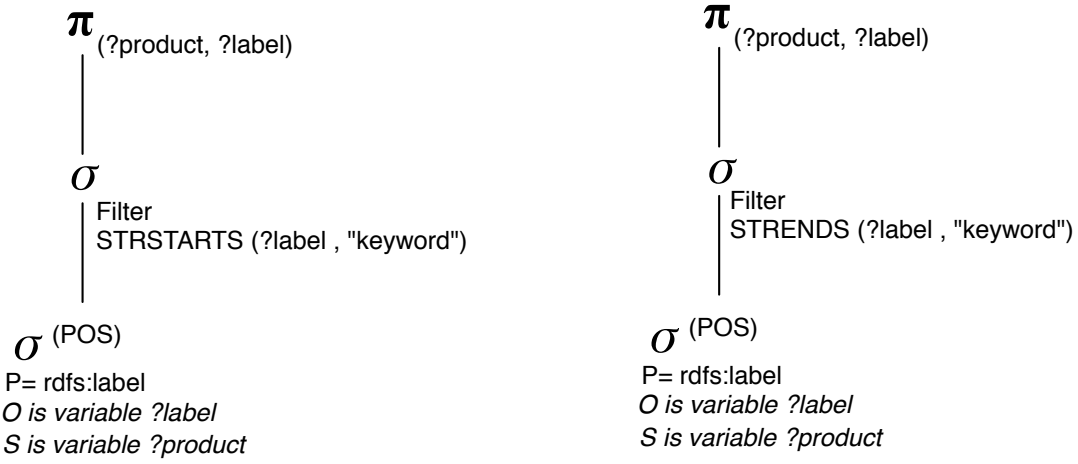


Figure 2.11: Expression tree for reducing Regex into XPath functions.

Physical Optimisations:
Physical optimisations of Regex have gained more attention, in general. Mainly, by building gram indices to reduce the result set to be scanned within the Regex engine, Cho and Rajagopalan (2002) proposed (FREE), a multi-gram index for a set of documents to be searched within a regular expression engine.

Lee et al. (2011) adopted FREE to be used within SPARQL through their system RegScan. FILT has also been proposed by Stuhr and Veres (2013) which builds a Lucene index, not just for evaluating Regex but also other logical filters.

2.4 Conclusion

The issue of Regular Expressions performance with SPARQL is not a straight-forward problem. In this chapter, a review of three main related topics (Data Base Management Systems, String matching problem and the Semantic Web) has been covered. SPARQL and RDF, as part of Semantic Web technologies, adopted the notion of triples which, although triples are relatively flexible, has been argued to be unpredictable. It is very hard to speculate how triples would be queried. For a given query, it is likely to have more joins within SPARQL than having the same joins within a SQL one.

In addition to this, Semantic Web technologies are still immature compared to the RDBMSs. This means that, in term of performance, RDBMSs is way a head of Semantic Web technologies. Moreover, Regular Expressions, as part of String matching problem, are also highly expressive, yet they are computationally expensive. SPARQL standardisations have proposed that Regex can be used to search through strings within SPARQL. This decision has put more pressure on the performance of different RDF stores, as part of BDMSs.

The major issue was that the SPARQL standardisations has chosen Regex to be the main string matching utility to be used within SPARQL. Yet, Regex is computationally expensive especially if it is to be used for simpler tasks (such as: simple string matching). The next chapter will attempt to investigate if this is the case.

Chapter 3

Regex Usage in the Context of SPARQL

To better understand Regex in the context of SPARQL, it is rather important to start by analysing how it is used by people. As Regex has a wide range of usages, the analysis may show cases where Regex could be executed more efficiently. The aim of this chapter is to identify those cases which will affect the design of the following benchmarks and the proposed index. This chapter attempts to answer the research question (1):

What are the trends/patterns that can be identified in people's usage of Regex within SPARQL?

In the next section, we re-introduce the two main concepts of this chapter, which are SPARQL and Regular expressions. Section 3.3 discusses our methodology regarding this chapter and how we approach it. We introduce the different datasets and their characteristics, and other statistics regarding our study. Section 3.4 has our different analysis strategies, which include dataset differences in Section 3.4.1, Regex pattern features and operators features that have been used in Section 3.4.2, Regex usage on the level of BGP in Section 3.4.3 and a comparison of the usage of both Regex and full-text search within DBpedia in Section 3.4.4, which can be seen as a comparison between exact and approximate matching within SPARQL. We deliver some suggestions at the end in Section 3.5.

3.1 Introduction

The SPARQL query language (Seaborne et al., 2013) allows queries of RDF data to be specified through Basic Graph Patterns (BGP). However, some usages also require the ability to search within literal values; to this end SPARQL provides the ability to filter values by matching against a Regular Expression (Regex).

However, matching Regex against large corpora (for example, the literal values in a large RDF dataset such as DBpedia) can be computationally expensive; it may not be possible to hold all records in the memory, which may result in expensive disk accesses. As a consequence, many SPARQL engines do not provide efficient Regex queries but, instead, provide conventional Full-Text Searching (FTS), often using existing text search engines, such as Lucene. However, SPARQL does not provide a standard syntax for querying FTS, so different SPARQL engines adopt different syntaxes/extensions.

In this study, we aim to explore the usage of Regex within SPARQL. A new trend in SPARQL query mining is by to analyse log files published by some dataset providers. These logs contain the actual queries that have been posed on their SPARQL endpoints; they may contain written SPARQL queries or queries generated by agents. DBpedia benchmark queries have been chosen, based on the analysis of real queries from DBpedia logs (Morsey et al., 2011). This style of analysing SPARQL log files has been the centre of the establishment of the Usage Analysis and the Web of Data workshop (USEWOD)[1], a dedicated workshop that was established in 2011 to study the use of queries on the Semantic Web.

3.2 Related Work

In the original SPARQL specifications (Prud'Hommeaux and Seaborne, 2008), querying strings was undertaken only through Regex, whereas SPARQL 1.1 (Seaborne et al., 2013) introduces a number of additional functions that are mainly related to XPath functions, namely STRSTARTS, STRENDS, CONTAINS, STRBEFORE and STRAFTER, which all can be seen as special cases of REGEX. Although FTS has been addressed within the working group[2], it has not been standardised so far; probably because there is no a common definition for the text search language. Moreover, implementing such a service may add technical constraints, whereas most triple stores usually reuse existing solutions as a preferred option, as Seaborne, one of the SPARQL 1.1 editors, suggested[3]. For example, Jena[4], OWLIM-SE[5] and Sesame (Minack et al., 2008) all use Lucene. Others use their own implementations. For instance, BigData implements a `B-Tree` full-text index[6]. Also, 4Store supports tokenising, double metaphones, and stemming as their FTS features, by implementing a unique forward chaining that stores the FTS data in RDF format[7].

[1] http://data.semanticweb.org/usewod/

[2] http://www.w3.org/2009/sparql/wiki/Feature:FullText

[3] http://mail-archives.apache.org/mod_mbox/jena-users/201306.mbox/%3C51C57EC0.2030601@apache.org%3E

[4] http://jena.apache.org/documentation/query/text-query.html

[5] http://owlim.ontotext.com/display/OWLIMv43/OWLIM-SE+Full-text+Search

[6] http://www.bigdata.com/bigdata/docs/api/com/bigdata/search/FullTextIndex.html

[7] http://4store.org/trac/wiki/TextIndexing

Regular expressions are a long-standing and well-studied topic in theoretical computer science; for an in-depth account of the state of the art and current research issues, see (Ellul et al., 2004). Most modern programming languages such as: Java, Perl, python and others, implement different techniques that are usually referred to as extended Regex or backtracking. Backtracking-based engines (flavours) usually have more features than what is offered by Finite State Machine (FSM)-based engines. For example, back referencing, as one of the new features provided by extended Regex engines, cannot be implemented within an FSM, as it is considered to be outside the regular languages (Becchi and Crowley, 2008). Using extended Regex engines, however, introduces new issues. For instance, consider the Regex pattern "(a+a+)+b" when matched against the string "aaaaaaaab". This pathological Regex, sometimes referred to as "Catastrophic Backtracking", can take an exponential time $O(2^n)$ and is considered to be an NP-complete problem (Becchi and Crowley, 2008; Reidenbach and Schmid, 2011). Using FSM-based engines, however, can solve such a pathological Regex linearly to the size of the input string as $O(n)$.

3.3 Methodology

To investigate how people are actually using Regex, we studied real-world queries that have been logged by SPARQL endpoint providers. We fed the queries into a SPARQL parser and perform different analysis regarding the usage of Regex, such as the usage of Regex pattern features. This analysis helped to identify issues within the usage of Regex. Details about the result outputs are included within Appendix A.

3.3.1 Research Data

Our study refers to the USEWOD research data set from the years 2013 and 2014. The collection comprises two formats: mainly, Common Log Format (CLF) logs, whereas the other small portion is in a plain text format. The collection contains queries that have been posed to different SPARQL endpoints. Table 3.1 shows a general description about each dataset; their triple store, the kind of Regex engine, the number of triples[8] and the period covered by those queries. Queries are collected on different dates within the period.

The log files originally contain SPARQL queries that have been posed against different SPARQL endpoints. Before analysing these queries, we give general statistics about the endpoints themselves and their contents. Table 3.1 lists each endpoint, its domain, their RDF store which should also implement a Regex engine which is also listed, how many RDF triple. Finally, we give the sample period that covers the collected log files.

[8] `http://stats.lod2.eu/rdfdocs?sort=triples` - [Accessed on: 01/03/2017]

Dataset	Domain	Store	Regex engine	Triples	Sample period
DBpedia	Cross	Virtuoso	Henry Spencer[9]	$\approx 325m$	01/07/2009-27/01/2014
SWDF	Bibliographic	Sesame2	Java.util.regex	246510	01/11/2008-22/01/2013
LGD	Geo-spatial	Virtuoso	Henry Spencer	$\approx 226m$	23/05/2011-12/01/2014
Bio2rdf	Medical	Virtuoso	Henry Spencer	$\approx 371m$	16/04/2011-25/06/2011
OpenBioMed	Medical	Jena TDB	Java/Xerces	883000	07/02/2011-19/11/2012
BioPortal	Medical	4Store	PCRE	$\approx 203m$	19/07/2012-12/12/2013

Table 3.1: Log files datasets and their general characteristics

These information are general in the field of SPARQL endpoints, but they also contain critical information about Regex such as (Regex engine) which may explain the Regex capability of different RDF stores. For example, Endpoints/RDF stores which have a Henry Spencer Regex engine type will not be able to perform a back-reference Regex queries (See Table 3.4).

Table 3.2 is the main table regarding Regex queries contained within the log files. For each dataset, it lists how many overall queries that is found within the log files. Then, it also shows the number of SPARQL queries which contains a Regex filter. It is also of important to analyse the length of those Regexes as this is a major factor that affects the performance of executing such queries. Generally, long Regex patterns may cause performance issues. In general, it is important to find more Regex queries/pattern for this research to be able to cover a representative Regex queries against those datasets. Those queries will be used across the rest of chapters in this thesis such as: designing a new Regex benchmark as in (Chapter 5). Finally, Table 3.3) is an attempt to show some Regex patterns sorted by their occurrences. It gives a general insight about those Regex queries.

The three tables: 3.1, 3.2 and 3.3 shed light on some characteristics about the usage of Regex within SPARQL. For example, DBpedia is particularly interesting; it is the largest dataset in terms of the number of triples 325 million triples as shown in Table 3.1, and it provides the largest number of queries in log files by around 49 million queries which is around 45% of the overall queries. DBpedia queries also contains the largest Regex queries 1.6 million Regex queries as 88% of the overall Regex queries contained in the log files as in Table 3.2. Those Regex queries seems to be posed by agents as half Regex queries from DBpedia comes from only two redundant patterns as shown in Table 3.3.

3.3.2 Processing Method

To extract Regex clauses from the log files, we use a SPARQL parser, namely Jena ARQ (Carroll et al., 2004). In theory, SPARQL is standardised and different triple stores are

[9] `http://www.arglist.com/regex/`

Dataset	Overall queries	Regex queries	Regex ratio	Regex length		
				Average	StdDev	Maximum
DBpedia	48,648,011 (45%)	1,623,710 (88%)	3.34%	51	27	224
SWDF	27,901,111 (26%)	64250 (4%)	0.23%	16	30	2,261
LGD	3,929,693 (4%)	133,192 (7%)	3.39%	9	5	236
Bio2rdf	192,081 (0.2%)	14 (0%)	0.01%	143	110	319
OpenBioMed	883,376 (0.8%)	281 (0%)	0.03%	11	6	43
BioPortal	26,375,686 (24%)	18,293 (1%)	0.07%	6	6	54
Total	107,929,958 (100%)	1,839,740 (100%)	%1.70	46	29	2,261

Table 3.2: General/Regex statistics about the datasets

Count	Ratio	Regex patterns	Dataset
616728	38%	`.+/((Company)|(Business)|(Company.*)|(CompaniesBasedIn.*)|(.*CompaniesOf.*))$`	DBpedia -
199394	12.3%	`.+/((Place)|(PopulatedPlace)|(Town)|(City)|(.*CitiesIn.*))$`	DBpedia -
8297	6.2%	`United States`	LGD
5342	4%	`Italy`	LGD
3619	5.6%	`(label|summary|name)$`	SWDF
1676	2.6%	`^http://data\.semanticweb\.org`	SWDF
7886	43.1%	`interaction`	BioPortal
991	5.4%	`rdfs:subClassOf`	BioPortal
197	70.1%	`^CG[0-9]*$`	OpenBioMed
29	10.3%	`concatenate`	OpenBioMed
2	14.3%	`drug`	Bio2rdf
1	7.1%	`el estring que quieras`	Bio2rdf

Table 3.3: Most often used Regex for each dataset and their occurrences

supposed to parse SPARQL equally. However, this is not always the case as different SPARQL engines may have their own extensions to the language, such as full-text search or spatial syntax. Table 3.2 shows the total number of all queries (not just Regex) for each dataset, the number of queries containing a Regex filter, the ratio of Regex queries against the overall queries and both the average and maximum length of Regex patterns. Only queries that have a Regex filter are parsed. From those, there are 295,671 invalid queries because of syntactical errors on the level of the SPARQL query. Those are not included within Table 3.2 as Regex queries. The length of a Regex pattern represents the entire pattern which includes both normal and meta-characters. For instance, the length of the pattern "(*type*|*class*|*subject*|*broader*)" is 28. Table 3.3 shows the most two frequently used patterns for each dataset and their occurrences. The source code is published on GitHub[10].

[10] `https://github.com/SaudAljaloud/SparqlLogAnalysis`

3.4 Analysis

In this section, we analyse the usage of Regex from different perspectives. Firstly, we analyse Regex in terms of the usage of Regex within datasets. Then, we examine the actual Regex patterns and their characteristics. We also examine Regex variables in the context of BGP and whether they are literals or URIs. Finally, a comparison between Regex and FTS is presented within DBpedia dataset.

3.4.1 Datasets Usage of Regex Filters

It is notable that the overall queries from DBpedia comprises 45% , the reminder being the other datasets combined. On the other hand, 88% of Regex queries are coming from DBpedia. Moreover, the average length of Regex patterns within DBpedia is longer than other datasets, as shown in Table 3.2. This might be because of the diversity of what DBpedia contains within the literals, or might be because of the structure of DBpedia itself, as it contains a large portion of free text. For example, "dbpedia:abstract" property can take up to 500 words of free text as an editorial limit when extracting data from Wikipedia (Bizer et al., 2009b). This gives an advantage to DBpedia to be further analysed in terms of Regex usage, as Regex is primarily affected by the length of the matched string.

Bibliographic (SWDF) and biomedical (Bio2rdf, OpenBioMed and BioPortal) datasets cover about 25% each of the overall queries, leaving only 4% for the Geo-spatial (LGD) dataset. However, Regex from the latter dataset comprises 7%, whereas the Bibliographic dataset only combines 4%, and 1% of the other biomedical datasets.

By comparing each dataset's overall queries against their Regex queries, we can see that DBpedia and LGD datasets have almost the same ratio of Regex, at 3.3%. The others show lower ratios: 0.2% for the SWDF and 0.07% for the biomedical datasets combined. However, by looking at Regex patterns from LGD, it is clear that most of the Regex queries are generated by an agent. As they are generated by an agent, the queries tend to have the same structure. Specifically, their Regex patterns do not contain any meta-characters. They are rather countries names corresponding to properties 8 and 9 from Table 3.5. Thus, analysing such a dataset may not provide a rich usage of Regex.

This leaves the only domain that contains more than one dataset within the log files; namely the Bio2rdf, OpenBioMed and BioPortal datasets. Biomedical datasets, here, have shown a general low rate of Regex so far. Despite the fact that 25% of the overall queries come from these datasets, Regex queries were only 4% and the ratio was 0.07%. Bio2rdf and OpenBiomed have only 13 and 14 Regex queries, respectively, after removing duplicates. In general, this might be the structure of such a domain that their datasets mainly consist of ontology classes rather than instances, as in DBpedia. Another reason

could be the constrained user interface for searching rather than writing queries through SPARQL endpoints.

3.4.2 Regex Operators/Features Analysis

In this section, we analyse Regex usage according to the use of Regex operators/features. However, Regex has many different flavours and features. We follow, in both definitions and terminologies, SPARQL specifications, which rely on the XQuery 1.0 and XPath 2.0[11] Regex syntax. XQuery and XPath also follow XML schema[12] with some additions, such as Start-with and End-with. The listed features are known to be supported by the investigated endpoints, see Table 3.1.

Stats/Features/Properties	Syntax/Description	Count
Not-well formed Regex	Does not contain "normal characters"	10,533
Case insensitive	flag "i"	1,551,138
Quantifiers:	*, + or ?	1,082,558
*	* (only *, not preceded by dot)	90,692
+	+ (only +, not preceded by dot)	1,250
.+	.+ (longest possible match, one r more)	952,724
.*	.* (longest possible match, zero or more)	854,203
Restricted quantifiers	{n,m}	1,392
Reluctant quantifiers:	??, *? or +?	1,000
.+?	.+? (shortest possible match, one or more)	0
.*?	.*? (shortest possible match, zero or more)	0
Character class:	[characters]	11,849
Negation	[^characters]	2,560
Grouping:	(characters)	1,014,720
Back reference	(character) followed by \1 or \2, etc	0
Alternation	Ex: character|character	1,031,389
Start-with	^one or more character	68,610
End-with	One or more character $	969,035
URIs	regex that start with http	329,383
Non-ASCII	contains chat >= 128	12,455
Exact search	normal characters surrounded with ^ and $	9,424
Surrounded by .*	In the form of .*keyword.*	1,083
Starts with .*	In the form of .*keyword	2,080
Ends with .*	In the form of keyword.*	26,295
Meta-characters free:	Patterns without any meta-characters	393,372
Just letters	only letters and/or white space	267,785
Just numbers	only numbers and/or white space	963

Table 3.4: Regex Stats/Features/Properties and their occurrences

Simple search properties, such as (*Start-with* and *End-with*) comprise almost half of all Regex clauses as Table 3.4 indicates. These properties are more index-friendly in

[11] http://www.w3.org/TR/xpath-functions/#regex-syntax
[12] http://www.w3.org/TR/xmlschema-2/

contrast to the other operations such as *Quantifiers*. Building an index for them kind can improve the speed of the search by an order of magnitudes, as proposed by Cho and Rajagopalan (2002); Lee et al. (2011). Moreover, SPARQL 1.1 has just added additional properties, such as STRSTARTS, STRENDS, CONTAINS, STRBEFORE and STRAFTER. SPARQL 1.1 differentiates between these properties/functions from Regex. Building dedicated indices for such properties will then introduce a trade off between the space and time.

It is also notable that there are no back reference Regex queries. This might be because back referencing is more expressive than is needed by users within the data searched by SPARQL. Back referencing is the only feature that cannot be implemented using FSM-based engines. It is probably that back reference is an advanced Regex feature that most users do not require for their filtering expressions. This also gives a hint about implementing FSM-based engines, as classical engines are claimed to be more efficient (Becchi and Crowley, 2008).

Regex is independent from human languages. This makes it sometimes the only solution when a FTS does not support a particular language. In this analysis, there are 12,455 Regex patterns that contain one or more Non-ASCII characters.

There are 9,424 exact search queries. Despite that this kind of Regex, where a start-with is presented within the pattern, may yield a best case scenario in terms of checking each record, this still requires a full scanning of records. A better way of writing such a pattern may be by not using a Regex filter; a direct query pattern that includes the full string could be more efficient, since SPARQL engines can benefit from their own data indices.

Just letters, just numbers and "Meta-characters free" mainly refer to the case where a Regex filter may not be the best way of writing a query. These may fall into the CONTAINS property introduced by SPARQL 1.1. However, these comprise around 20% of Regex queries.

Regex is not just for searching string literal, but also can be applied on URIs. Searching URIs comprises 20% of all Regex queries. For example, "http://dbpedia.org/ ontology/" is used 81,390 times as a Regex pattern. From the list of URIs we analysed, there is a common style for these queries usually associated with prefix queries, such as "http://dbpedia.org/*", which has 51,851 occurrences. It is probably the case where the desired pattern is actually "http://dbpedia.org/ontology/.*". This is a common mistake users make by using "*" as a wildcard syntax. However, it is interpreted as a Regex quantifier and, therefore, produces different results. Table 3.4 indicates that "*", not preceded by a dot, is used 90,692 times.

ID	Count	Predicate URI	Range
1	1,024,422	`http://www.w3.org/1999/02/22-rdf-syntax-ns\#type`	rdfs:Class
2	123,684	`http://dbpedia.org/property/name`	rdfs:Literal
3	87,153	`http://www.w3.org/2000/01/rdf-schema\#label`	rdfs:Literal
4	69,257	`http://www.w3.org/2000/01/rdf-schema\#subClassOf`	rdfs:Class
5	39,249	`http://linkedgeodata.org/ontology/directType`	rdfs:Class
6	29,559	`http://xmlns.com/foaf/0.1/name`	rdfs:Literal
7	23,822	`http://dbpedia.org/property/reference`	rdfs:Literal
8	17,097	`http://linkedgeodata.org/property/is_in`	rdfs:Literal
9	12,959	`http://linkedgeodata.org/property/is_in\%3Acountry`	rdfs:Literal
10	12,515	`http://xmlns.com/foaf/0.1/page`	foaf:Doc

Table 3.5: Most often used predicates, where their objects are regexed

	Overall queries	FTS queries	Regex queries	FTS AND Regex queries
DBpedia	48,648,011	169,204	1,672,890	18,554

3.4.3 Regex in Relation to BGP

In SPARQL specifications, Regex is written within a filter expression. Moreover, there has to be a variable that is referred to from the BGP to be searched with Regex. We analysed Regex variables in relation to the BGP. There are 74,708 of subject variables, 41,636 predicate variables and 1,507,304 object variables that have been matched with Regex. Although it is expected that objects are the most used as Regex variables, as they only can hold literals in terms of RDF semantics, subjects and predicates still can be used for searching URIs or blank nodes. In this special case where the referred Regex variable was an object, we also analysed the predicates of that object. Examining these locations can help in determining whether the scanned variable is actually a literal or a different type, such as URI or a blank node that has been translated into a string literal (using STR function), for the purpose of searching these variables using Regex. Moreover, identifying the predicates where their objects are more likely to be searched with Regex, can help in building indices for such predicates, as the case in Jena-Text[13] where a pre-defined predicates can speed up the search for FTS. Table 3.5 shows the top 10 predicates/properties, which have their objects searched within Regex.

3.4.4 Regex Vs FTS within DBpedia

DBpedia has shown a rich Regex usage so far. Moreover, DBpedia has also maintained a FTS extension with their "bif:contains" property. We aim to examine how many FTS queries have been used and to compare them with the Regex ones. Most importantly, how many queries contain both, and whether they are written optimally.

In the cases where both FTS and Regex are within one SPARQL query, there are a large portion that have the same SPARQL structure. This most probably refers to the

[13] `http://jena.apache.org/documentation/query/text-query.html`

agent that submits these queries. For example, the SPARQL query 3.1 shows a good integration of FTS and Regex. In this query, instead of scanning all records within Regex, a FTS index will produce a subset of candidates to only be filtered with Regex. However, there are some issues that can be addressed from this query; mainly regarding false-negative results. In the case of Regex, a user should have an understanding of what results are expected. For example, a label, such as "East India Company" is not going to be returned, as India is not at the start of the line. On the other hand, a user should also have an understanding of how FTS indexing works. For example, it is not always the case that the FTS tokeniser implements stemming; in this case, a label like "Indian Railways", for example, is also not going to be returned, as "Indian" would not be matched by the FTS and, therefore, it will not be filtered with Regex.

```
SELECT DISTINCT ?s ?o WHERE {
 ?s <http://www.w3.org/2000/01/rdf-schema#label> ?o .
 ?o bif:contains "India".
 FILTER (regex(str(?o), '^India')) .
 FILTER (!regex(str(?s), '^http://dbpedia.org/resource/Category:')).
 FILTER (!regex(str(?s), '^http://dbpedia.org/resource/List')).
 FILTER (!regex(str(?s), '^http://sw.opencyc.org/')).
 FILTER (lang(?o) = 'en').
}
Limit 10
```

Listing 3.1: SPARQL query example.

3.5 General Remarks

Our analysis showed that there are issues of how Regex being used by different bodies. In addition, it was expected that Regex would be mostly applied to literal strings (human-readable labels, such as rdfs:lable), but it was mostly on filtering URIs. Based on our findings, we deliver a number of general suggestions that can help to increase the efficiency of Regex queries within SPARQL:

- Users/SPARQL engine developers may take into account rewriting Regex queries to be integrated with a FTS extension.
- W3C may consider adding additional specifications regarding Full-Text Search within SPARQL. This can help users to differentiate between FTS and Regex, and to use the appropriate one.
- SPARQL engine developers may build indices for some of the most used properties, such as START-WITH, END-WITH and CONTAINS, as they already have been syntactically added to SPARQL 1.1 specifications.

- SPARQL engine developers may consider adding a FSM-based engine to be used within simple patterns.

3.6 Conclusion

Regular Expressions are a powerful technique that can be used for matching string data. Although they maybe computationally expensive in some cases, they have a high degree of expressivity. For example, they can be used to match simple strings, URLs, phone numbers, back-referencing and many other cases. In this chapter, we have studied the issue by analysing how users/agents are using Regex within SPARQL. This chapter indicates that there are various patters that people use with Regex and SPARQL that can be made more efficient. One of the main findings is that people are actually using Regex to perform a full-text search. However, the SPARQL standardisation does not specify a way to this utility and therefore it is being either used with Regex by people which will cause a performance delay. The other possibility is that RDF store vendors may have their own implementation of full-text search which preaches the idea of standardisation.

Studying how people are using Regex within SPARQL (while writing their queries) by analysing the log files is still not enough to understand the actual performance of Regex within SPARQL. It is equally important to empirically assess the performance of Regex within SPARQL when executed against some RDF stores which is investigated more in the next chapter.

Chapter 4

BSBMstr: Benchmarking Regex and XPath within SPARQL

Having investigated the actual usage of Regex within SPARQL by people, another question has to be answered (2):

What are the factors that empirically affect the performance of Regex within SPARQL?

In this chapter, we propose BSBMstr, a Regex and XPath benchmark that aims to assess the performance of Regex within SPARQL across different RDF stores. For those who expect to have many Regex queries over SPARQL, this benchmark can help in choosing the right RDF store. We study the performance of Regex within SPARQL through extending the Berlin SPARQL Benchmark (BSBM). We also investigate what causes the slowness of Regex in an empirical fashion by carefully designed queries, which allow us to compare certain queries with others in different forms. Our contributions can be summarised as follows:

- Investigation of the factors that affect the performance of Regex within SPARQL, and
- Provide a comprehensive search mechanism through strings using the Regex and XPath extension benchmark across seven popular RDF stores.

The next sections are as follows: Section 4.1, which discusses the related work in terms of benchmarking and Regular expressions. Section 4.2, which is the main section where we describe the extension of BSBM and the rationale behind our chosen queries for both Regex, in Section 4.2.1, and XPath in Section 4.2.2. Within Section 4.2.4, we analyse Regex and XPath separately through two main metrics: QMPH and QPS. In Section 4.2.6, we compare Regex to XPath queries where they are comparable. Finally, in Section 4.3, we conclude with some lessons that have been learnt from this benchmark and reach a conclusion stated at Section 4.4.

4.1 Related Work and Discussion

Benchmarking is one of the main topics that draws attention not just in relation to RDF storages, but also for "computer systems" in general through the well-known Transaction Processing Performance Council [1] (TPC) benchmarks. Google proposed Octane[2], which is a JavaScript performance benchmark for the modern web; Regular expressions are one of the main factors they measure on the client side. As RDF BDMSs are starting to evolve, benchmarks have become important to measure the general performance and scalability under various workloads or throughputs, such as the Berlin SPARQL Benchmark (BSBM) (Bizer and Schultz, 2009), the Lehigh University Benchmark (LUBM) (Guo et al., 2005), the DBpedia SPARQL Benchmark (Morsey et al., 2011), SP2Bench (Schmidt et al., 2008), and others[3]. In some cases, as SPARQL has a wide range of features which can affect the general performance, some specific benchmarks have been proposed, such as FedBench (Schmidt et al., 2011) or FedX (Schwarte et al., 2011) for benchmarking federated queries. Other benchmarks have been introduced for testing even unsupported features or extensions of SPARQL[4], such as LUBMft (Minack et al., 2009) for benchmarking full text queries by extending the LUBM benchmark or Geographica for benchmarking geo-spatial queries (Garbis et al., 2013) or C-SPARQL (Barbieri et al., 2009) and SRBench (Zhang et al., 2012) for streaming RDF data.

However, some benchmarks, where their data is synthetic, have been criticised in that the data does not reflect real world usages and, therefore, will not show a fair comparison (Duan et al., 2011). Mostly, the structure of the data has been claimed to be relational-style schema, which is not the case for realistic RDF datasets. This may have some effects on queries that require expensive joins. In our approach, we aim at focusing on Basic Graph Patterns (BGP) within a single triple pattern, meaning no joins would be performed. The Linked Data Benchmark Council (LDBC) (Angles et al., 2014), is a dedicated project that is meant to bring industry and academia together to develop better benchmarks and RDF datasets. LDBC indicates the increased importance of benchmarking RDF and SPARQL in the area of the Semantic Web.

Regular expressions, however, have not been adequately studied within RDF benchmarks or within SPARQL, in general. Most studies have investigated Regex for path traversal between nodes as in (Alkhateeb et al., 2009; Losemann and Martens, 2013). We, however, focus on benchmarking Regex for matching literal strings through the Regex filter expression within SPARQL. Lee et al. (2011) proposed (RegScan), a multi-gram index to speed up Regex queries by reducing the result set to be scanned. The queries they used for evaluating their proposal were helpful in our query design, though not exclusively so.

[1] http://www.tpc.org/information/benchmarks.asp
[2] http://code.google.com/p/octane-benchmark/
[3] http://www.w3.org/wiki/RdfStoreBenchmarking
[4] http://www.w3.org/wiki/SPARQL/Extensions

Name	**Domain**	**Scope of queries**	**Include regex?**	**Features**
BSBM	E-commerce use cases	General	Was removed	%keyword%
LUBM	University domain	Extensional queries	NONE	NONE
SIB	Social networks	Graph queries	Include Regex	"keyword" .
DBpedia	DBpedia dataset	Real-queries	Include Regex	Exact, OR, URLS
FedBench	Multi-domain	Federated query	NONE	NONE
LODIB	Linked Data translation	Translation patterns	NONE	NONE

Table 4.1: Comparison of different benchmarks on their usage of Regex

FILT (Stuhr and Veres, 2013) also addresses the same problem, as well as some other filter expressions, such as numerical and logical expressions, by building a Lucene index. Their benchmark queries included the four SPARQL query forms: SELECT, DESCRIBE, CONSTRUCT and ASK. It is, however, unclear what Regex patterns were tested in FILT. Moreover, both RegScan and FILT respectively use relatively smaller datasets; 869,770 and 1,771,100 at most. RegScan evaluated their system, which is a C++-based prototype against Sesame, a Java-based framework for processing RDF data. We compare seven popular RDF stores to each other with a dataset of sizes up to 8 million triples.

Unlike FILT, In this chapter, the focus is to only test SELECT queries. This is because the intention is to investigate the performance of different Regex patterns within SPARQL (not SPARQL queries in general), given a relatively fixed SPARQL query structure to maintain consistency while testing. Queries will be compared to each other at some point in the analysis, and it is important to keep other variables (such as: query forms: SELECT, CONSTRCUT and so on) controlled apart from the Regex patterns variations. We have also excluded unnecessary SPARQL features such as: OPTIONAL, UNION, other FILTERS. These SPARQL query features will cause execution variation and therefore will make reasoning over the performance of Regex within SPARQL even more subtle.

Table 4.1 indicates that only a few benchmarks have included Regular expressions filters. In fact, in the case of BSBM, they removed Query 6 that had Regex from the last version 2013[5] and stated that "*This query would become very expensive in the 1G and larger tests, so its performance would dominate the result*" . As far as we are aware, there is no benchmark for searching literals over RDF stores using Regular expressions or other string functions, such as XPath. We aim to extend BSBM for a special Regex and XPath benchmark.

[5] `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/`

4.2 Extending the Berlin SPARQL Benchmark

The Berlin SPARQL Benchmark (BSBM) is designed to measure the performance of RDF stores with different architectures that expose SPARQL endpoints. It proposes: data generation, set of queries, test driver and metrics. The benchmark is built around e-commerce use cases, such as: explore and business intelligence. The queries from these use cases do not measure literal strings retrievals. There used to be Query 6 in the previous versions of this benchmark for Regex, but it was later dropped as it was causing a bottleneck with regards to other queries' speed, especially with the metric QMPH, which takes into account the average of executing all queries across all runs. Our extension (BSBMstr) can be seen as a new use case named a literal search use case.

In the queries below, we aim to reduce the size of the BGP to a single triple pattern. This can help in measuring the intended features rather than other parts of SPARQL, such as joins, which may have side effects on our results, especially as joins have major effects on the performance of RDF stores. In addition, they have already been studied and benchmarked with most of the existing benchmarks or other specific studies, such as (Neumann and Moerkotte, 2011; Huang et al., 2011).

The full list of Regex and XPath queries together with details of the output results are included in Appendix B.

4.2.1 Regex Queries

As Regex are part of SPARQL specifications, we use the same queries for all stores. Moreover, the Regex language has already been defined and agreed across different Regex engines, it is less likely to find differences in the number of returned results. In writing Regex patterns, we follow SPARQL recommendations, in that it follows XPath/X-Query[6], which in turn, follow XML schema[7] with additions such as Start-with, End-with and back-referencing. Queries 1, 2 , 3 and 4 show how the length of data can affect the performance. Queries 4 and 7 show how ".*" affects the performance while retrieving the same results.

With this benchmark, it is not our intention to test yet another triple patterns order, such as star, chain or circle. Our goal is to measure the performance of Regex in isolation to other factors, such as joins across different RDF stores. Below, each of these queries contains only one triple pattern in addition to the filter expressions. They also do not include any other SPARQL syntactic features, such as DISTINCT, OPTIONS, UNIONS, ORDERBY or LIMIT. This, in addition to benchmarking the overall Regex performance of different RDF stores, can help to compare queries in what we intended to test, such

[6] http://www.w3.org/TR/xpath-functions/#regex-syntax
[7] http://www.w3.org/TR/xmlschema-2/#regexs

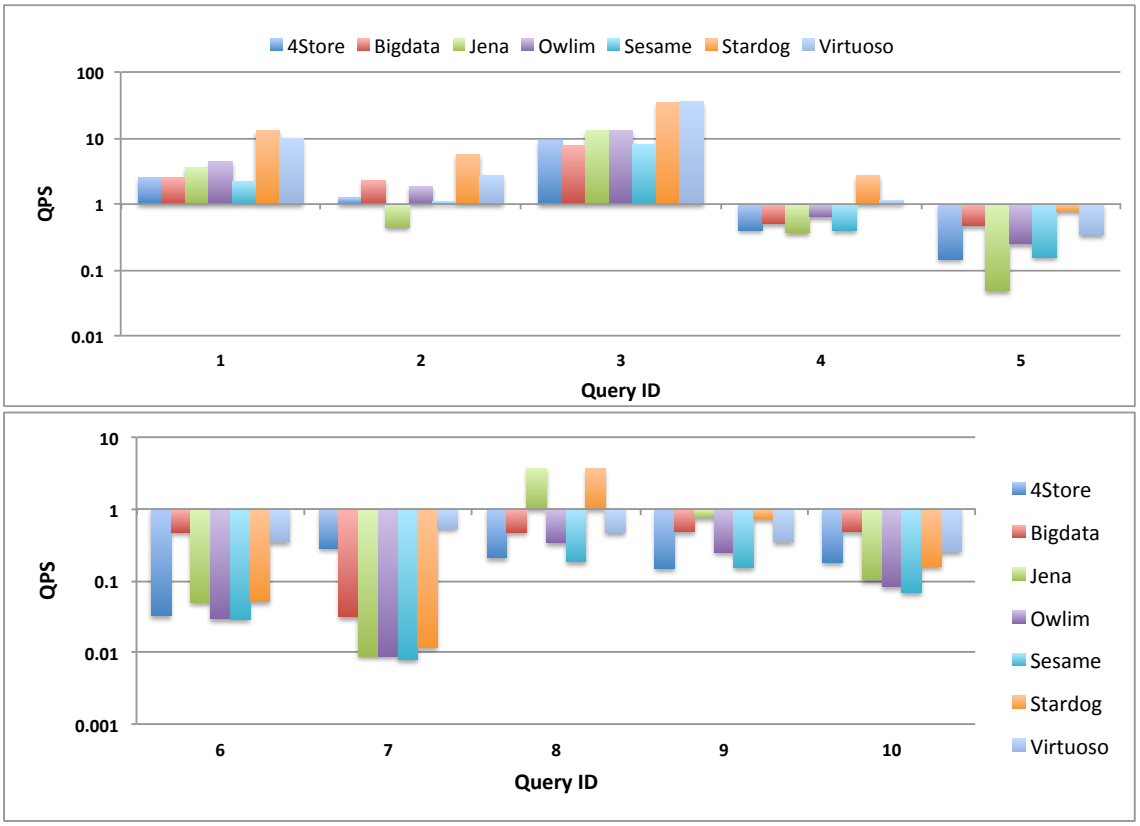


Figure 4.1: QPS of RDF stores on different Regex queries using 8M triples and 1 Client.

as literal length, cardinality and Regex complexity. Query 5 is the only query containing more than one triple pattern; meaning that this query performs a single join. Queries 6, 8 , 9 and 10 test how the Regex pattern itself affect the performance, while keeping the length of tested literal strings the same.

The performance of Regex, in general, is being well-studied in the area of theoretical computer science. The two main factors that affect Regex are: the complexity of the pattern itself as well as the length of the string being evaluated against. In this experiment, we took these two factors into consideration while writing the queries. The process of concluding the below queries is to cover as many Regex patterns as possible to be investigated. In addition to this, we are also interested to investigate how the length of string. For the Regex pattern complexity, we follow the SPARQL specifications in relation to Regex. Regex SPARQL specifications follow XPath/XQuery[8]. We examined their guidance and listed all Regex patterns features such as: simple Regex pattern which only contains letter without any meta-characters, qualifiers (such as * or +), Start-with and so on. Then, we merged these Regex patterns into simple SPARQL structure to minimise side effects in measuring the performance. For example, having an OPTIONAL clause within the SPARQL query will, in general, reduce the performance

[8] `http://www.w3.org/TR/xpath-functions/#regex-syntax`

a SPARQL query. The queries also include some special cases that we thought it have a worth investigating performance into it such as: Query 7.

To also investigate the length of string, we include various length triples to be scanned. Thanks to BSBM dataset, we are able to explicitly choose tables that we already know the length of their strings. For example, product labels are shorter than product comments. Reviews names are way shorter than reviews text, and so on. We use a combination of the two factors against Regex patterns. For example, Query 1 and 2 have the same Regex pattern complexity while having different set of triples to be scanned. This should show how the length of string will affect the performance of Regex queries within SPARQL.

Query 1: Find products whose labels include a particular term.
Query 2: Find products whose comments include a particular term.
Query 3: Find reviewers whose names include a particular term.
Query 4: Find reviews whose titles include a particular term.
Query 5: Find reviews whose texts include a particular term.
Query 6: Find reviews whose texts include any of three terms.
Query 7: Find reviews whose titles include a particular term surrounded by ".*" .
Query 8: Find reviews whose texts start with a term.
Query 9: Find reviews whose texts end with a term.
Query 10: Find reviews whose texts include a term repeated three times.

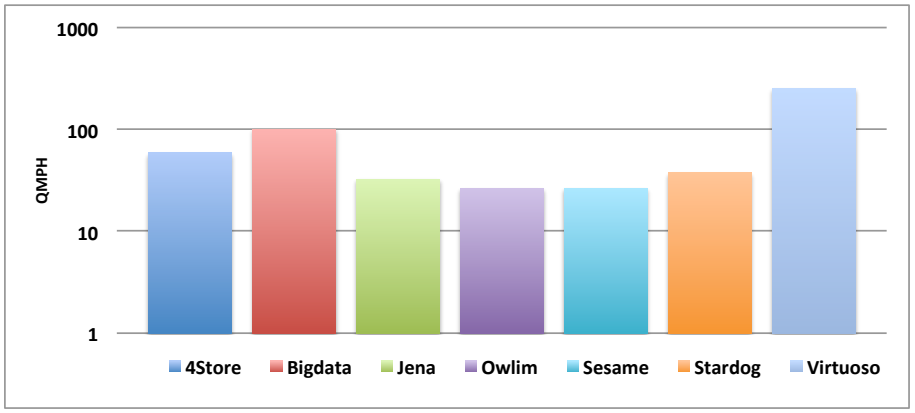


Figure 4.2: RDF stores comparison using QMPH against a set of Regex queries using 8 Million triples of BSBM dataset.

4.2.2 XPath Functions

This section includes three functions that have been added to SPARQL 1.1. The last query 4 shows a string equality search. This query is meant to show the performance difference of comparing the whole string to searching through one. It would also show

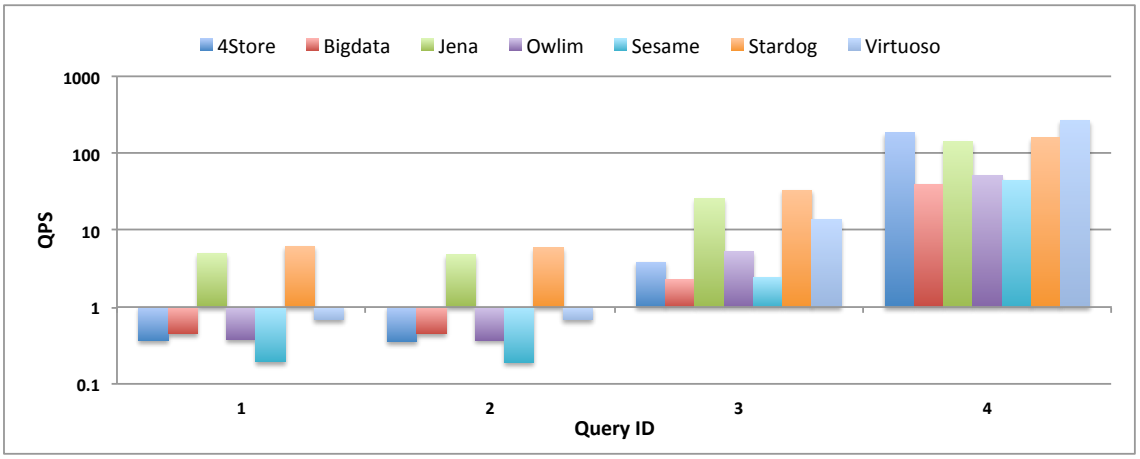


Figure 4.3: QPS of RDF stores on different XPath and direct string queries using 8M triples and 1 Client.

how RDF stores perform with such a query. Queries 1 and 2 are comparable to themselves and to Queries 8 and 9 from Regex. Query 3 is not comparable to Queries 1 and 2 as they do not test the same results. Query 3 is comparable to Query 1 from Regex.

Query 1: Find reviews whose text start with a particular term.

Query 2: Find reviews whose text end with a particular term.

Query 3: Find products whose labels include a particular term.

Query 4: Find products whose labels are equal to a particular term.

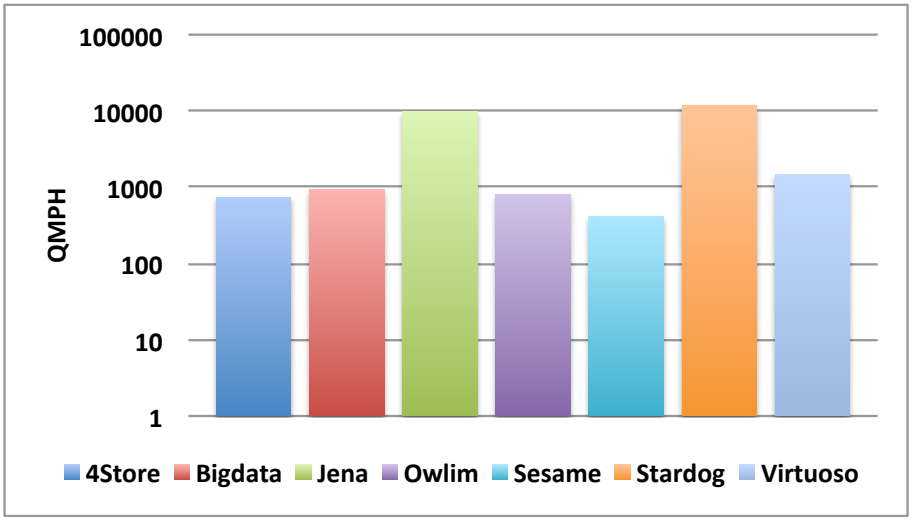


Figure 4.4: RDF stores comparison using QMPH against a set of XPath queries.

4.2.3 Experiment Design

We reused BSBM tools, such as metrics, dataset, test driver and procedures. In the case of the dataset, unlike BSBM, where datasets have the sizes of 100, 200 million triples,

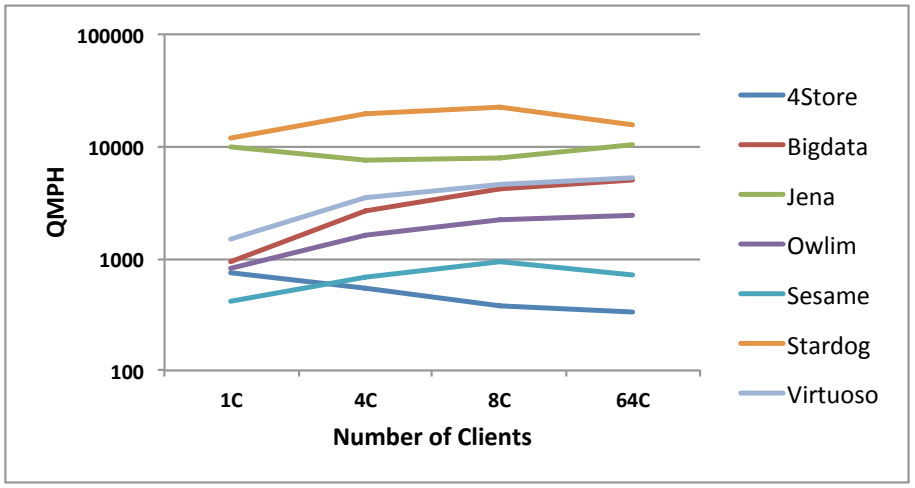


Figure 4.5: RDF stores comparison using QMPH against a set of XPath queries with multiple clients.

we only use 1, 2, 8 million triples. This is because Regex queries take longer to run. We also amended the number of warm up runs to 50 and the actual runs to 200. The BSBM version that we use is OpenLink BIBM Test Driver 0.7.7. We, however, introduce a new set of queries for testing both the Regex and XPath functions. We also extend the number of RDF stores from four, in the latest version of BSBM, to seven RDF stores, in our case. We use the latest versions as follows: 4Store: 1.1.5, Bigdata: 1_3_0 Revision 8501, Jena: 2.11.1 Fuseki: 1.0.1, Owlim-SE: 5.5 Tomcat: 6.0.39, Sesame: 2.7.11 Tomcat: 6.0.39, Stardog: 2.1.2, Virtuoso: Virtuoso Open Source Edition (Column Store) (multi threaded) 7.1.1-dev.3208-pthreads. For each store, we installed the latest version at the time of testing. Moreover, we contacted each of these stores' team to ask for the right configuration of their system to ensure fair comparison by giving them a general overview of the benchmark with a sample query. Then we describe the stages of setting up their store. We asked if there were any missing parts. Most of the stores replied and their suggestions have been taken into consideration where applicable. Only Stardog did not reply to emails, although their documentation is clear enough and we did not have any issues in setting up the store.

The machine we used to run our test is a dedicated machine that has the following specifications: 2x AMD Opteron 4280 Processor (2.8GHz, 8C, 8M L2/8M L3 Cache, 95W), DDR3-1600 MHz 128GB Memory for 2CPU (8x16GB Quad Rank LV RDIMMs) 1066MHz 2x 300GB, SAS 6Gbps.

4.2.4 Regex Analysis

There are two main metrics that have been used for this experiment: Query Per Second (QPS), Query Mix Per Hour (QMPH). Moreover, we intend to test a number of factors that affect the performance of Regex; these are data size and number of clients, across

seven RDF stores. It is clear that the data size is relatively smaller than the original BSBM which have: 100 million, 200 million and up to 1 billion triples in the case of Owlim-SE and Virtuoso. With a machine comparable to the BSBM machine, Regex queries are still performing slower.

4.2.4.1 Comparing RDF Stores on Regex Performance

Figure 4.2 gives a summary of the performance of all Regex queries across RDF stores using 1 client and 8 million triples, measured by the QMPH metric. Virtuoso outperforms others by 150% QMPH than the nearest Bigdata. In general, Virtuoso has been reported to be faster on different benchmarks/use case queries (Bizer and Schultz, 2009; Morsey et al., 2011). In terms of Regex, Virtuoso embeds a known Regex engine, the Henry Spencer algorithm[9] which is used by known BDMSs, such as MySQL[10] and PostgreSQL[11]. Sesame and Owlim-SE, however, both perform the lowest. In the case of Owlim-SE, it has also been indicated by Bizer and Schultz (2009) in its latest versions 2013 that Jena TDB outperforms Owlim-SE.

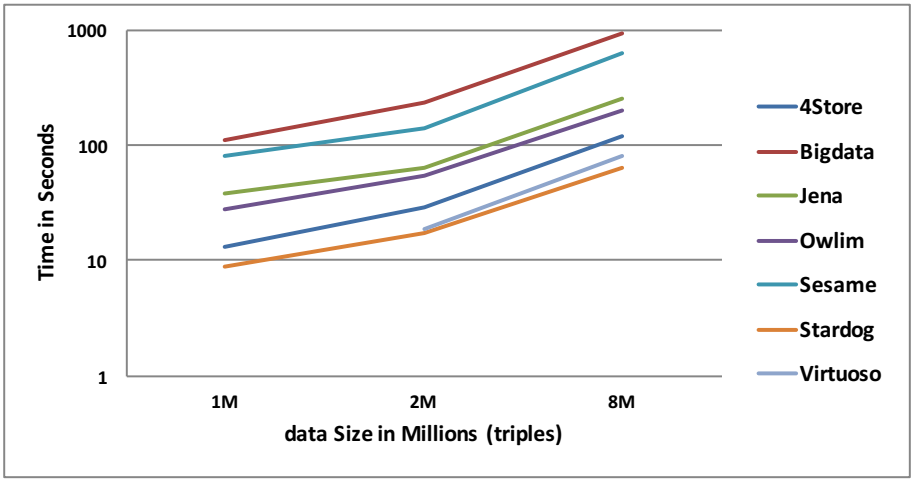


Figure 4.6: Loading time comparison of different BSBM dataset sizes across different RDF stores.

Figure 4.7 indicates the performance changes over RDF stores using different data sizes: 1, 2 and 8 million triples. In general, the size of the data has a proportional negative effect on the performance of Regex queries as sizes go up. This is because the complexity of Regex is mainly affected by the length; in this case, it is represented by the number of records added. Figure 4.8, however, takes another perspective, where we test how the number of clients sending Regex queries concurrently affects the general performance, using a data size of 8 million triples. All stores gain a positive effect when clients go from 1 to 4, by around 100%. On the other hand, going from 4 to 8 clients, has variant results.

[9] http://www.arglist.com/regex/
[10] http://dev.mysql.com/doc/refman/5.5/en/regexp.html
[11] http://www.postgresql.org/docs/8.3/static/functions-matching.html

Jena has a negative impact by 50%, but recovers and back to an average point with other stores when applying 64 clients. Others, such as 4Store, Bigdata and Sesame have no impact. In general, Virtuoso has gained the most benefits from using concurrency, by around 150% and 100% from 1 to 4 to 8 clients, respectively. This behaviour has also been encountered in the Berlin SPARQL Benchmark, where some stores do not benefit from concurrency at some points. Figure 4.6 depicts how long, in seconds, it takes each RDF store to load the different dataset sizes (workloads) of BSBM (1, 2 and 8) million of triples into their databases. On the other hand, Figure 4.9 shows the resulting size in megabytes of these databases after RDF data are being loaded into these databases.

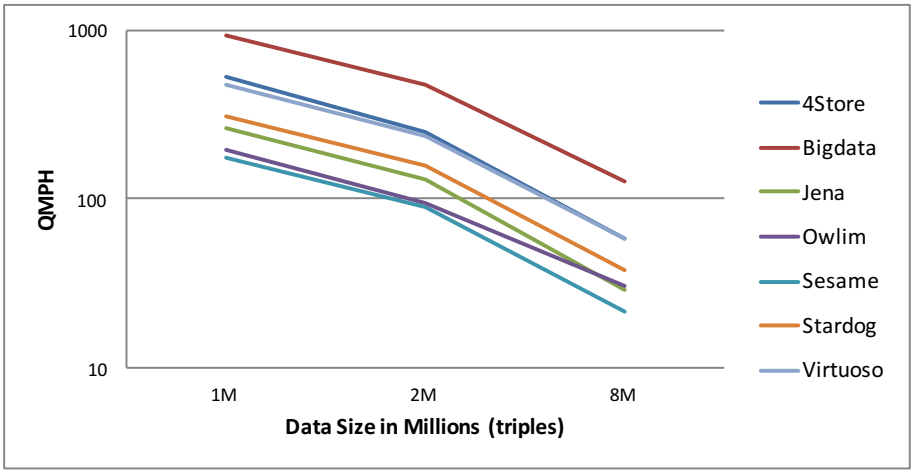


Figure 4.7: The overall performance of RDF stores using single client across different BSBM dataset sizes.

In summary, this test provided an overall comparison of the performance of different RDF stores by applying different factors. However, most of the results here were affected by the general performance of the stores, rather than being caused by Regex. The next section provides more detailed analysis of what actually affects the performance of Regex, regardless of which store is being used.

4.2.4.2 Regex Complexity Analysis: Empirical Approach

Figure 4.1 shows the performance of different Regex queries, using the Query Per Second (QPS) metric, meaning the larger numbers the better. Query 3 was the most efficient across all stores. This is because `foaf:name` has lower records and shorter length in relation to other predicates, such as `rdfs:labels`, `rdfs:comment` or `rev:text`. Query 7, however, was the worst, due to the two quantifiers at the beginning and the end. Quantifiers result in excessive backtracking to match the write string (Yang et al., 2011).

Queries 1, 2 and 4 showed how the length of strings affect the performance of Regex in a linear fashion, when using the same Regex pattern complexity (literal matching by only

one keyword) type. The tested predicates were: `rdfs:label`, `rdfs:comment` and `dc:title` which increased in length of strings, respectively[12] and, therefore, decreased the performance of Regex in the same manner.

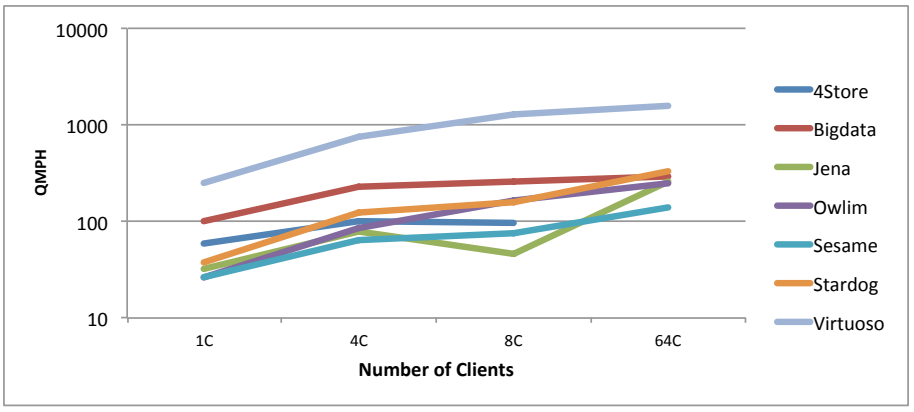


Figure 4.8: The performance of Regex queries against different clients using 8 Million triples from the BSBM dataset.

Queries 5 and 6 showed, on the other hand, how the complexity of the Regex pattern affects the performance of the query. In Query 5, the pattern was in a simple literal match, whereas Query 6 involved three alternations. This caused Query 6 to perform inefficiently, because of the Regex pattern complexity caused by long patterns with alternations. A mitigation may be gained by following an appropriate Regex design pattern which, for example, requires a pattern to be compiled once, but matched as many. We investigated most of the test stores and the only store which follows this design pattern is Jena, which in this experiment shows an identical performance across the two queries. Nevertheless, it is not enough to compile patterns once to gain a steady performance, as the complexity of the Regex pattern can still affect the matching process as well.

Furthermore, queries 7 and 4 were comparable to each other, as they tested the same set and returned the same result. However, all stores outperformed in Query 4 by more than an order of magnitude. In fact, patterns should be written as per Query 4, as all the tested RDF stores used a "find function" rather than an "exact function" one. The ".*" are actually used in many real-world cases, as investigated in Chapter 3, yet they should be written without appending ".*" which just adds unnecessary computations.

Queries 8 and 9 gave a comparison for commonly used cases about start and end with matching. In general, a start-with query 8 is faster than an end-with query 9. In Regex engines, simple start-with patterns, which do not involve other meta-characters, can be verified by fewer steps, making it faster to declare failure than still trying to match. On the other hand, end-with requires the engine to scan characters until the end in all cases, making it only able to declare either a match or failure at the end of the string.

[12] `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/Dataset/index.html`

Query 10 was a back-reference query that informed a couple of tests on the stores. It checked whether a store supports various meta-characters, such as grouping, classes, ranges, and most importantly back-referencing capabilities. It showed that all RDF stores actually support back-referencing and other meta-characters used by this query, since they all returned the same result sets. It also showed a relatively good performance on average, although back-referencing is non-regular, and causes exponential behaviour (Namjoshi and Narlikar, 2010).

In conclusion, this experiment shows that Regex is actually not only affected by the expensive disk accesses, but also by the complexity of Regex patterns. In this next section, we benchmark other simpler functions for searching through strings. These are mostly XPath functions, which show how simpler functions, other than Regex, can affect the performance of executing SPARQL queries across RDF stores.

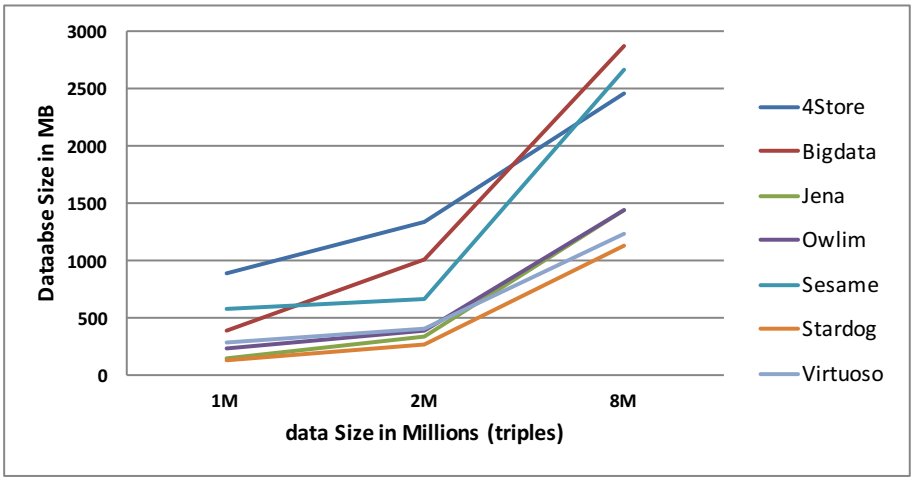


Figure 4.9: Database size comparison of different BSBM dataset sizes across different RDF stores.

4.2.5 XPath Functions Analysis

Regex used to be the only main tool that SPARQL specifications supported for searching through strings. Then, in SPARQL 1.1, three main additional functions were added, namely, STRSTARTS, STRENDS and CONTAINS, which are XPath functions. These functions can be seen as a subset of what Regex can be used for. These functions have important advantages. Firstly, they are less complex than Regex and, therefore, can be efficiently computed. Moreover, the study in Chapter 3 on the Regex logs showed that these functions comprised a large portion of what people are actually using Regex for. In this experiment, we tested the performance of these functions using the same standards that were used for testing Regex on 8 million triples.

Figure 4.4 illustrates the overall performance of stores, and shows dramatic changes from the Regex results in Figure 4.1. In this case, both Stardog and Jena TDB outperformed

the others by an order of magnitude. Others had relatively similar performances to each other, by around 1,000 QMPH.

Here, we discuss each query from faster to slower, in QPS, respectively, as presented in Figure 4.4. Query 4 performed the best over the other queries; this is because of the equality search that this query is executing. This query was the only query which did not involve a filter, but, rather, the string to be searched was specified within the graph pattern. This query is fast because most stores would directly traverse the `B-Tree` for finding the equal nodes in logarithmic time $O(log\ n)$. Some stores, such as 4Store, make use of additional hash tables which, when loaded into memory would perform searching in a constant time $O(1)$; there would not be any added computation on the string. Query 3 was not comparable to Queries 1 and 2, as they tested different sets of records. The reason that Query 3, which tested the CONTAINS function, was faster than even STRSTARTS was that `rdfs:label` had both a lower number of records and a shorter average length than `rev:text`. Both Queries 1, 2, which tested STRSTARTS and STRENDS, respectively, had identical performance across their RDF stores. These functions have the lowest complexity, where the engine only checks the desired characters; it is easier for such functions to declare failure over success, as they do not need to continue comparing characters. However, in this case, although the average length of the matched string did not affect the performance, the number of its records was relatively higher than, for example, `rdfs:label`. This was the cause of these two queries being slower than Query 3.

Figure 4.5 shows how this benchmark is affected by adding more than one client executing the queries concurrently. Stardog increased its performance from 1 to 4 clients by a factor of 2, then increased by a small portion with 8 clients, but declined with 64 clients. Jena had almost an opposite curve, with the same start and end as Stardog. Jena started with a slight decline from 1 to 8 clients by around 25%, then recovered at 64 clients, to be at the same performance at 1 client. Virtuoso and Bigdata were the ones which seemed to benefit from multi-threads. They improved by about a factor of five from 1 to 64 clients. They were also followed by Owlim-SE which had a steady curve over 4, 8, and 64 clients by a factor of three. Sesame was, however, affected by the 64 clients, which resulted in a slight decrease, marking the overall improvement by only around 80%. 4Store had a major drawback, whereby each time the number of clients was increased, the slower it became. This resulted in an overall decrease of around 55%.

4.2.6 Comparing Regex to XPath Queries

There are some queries between Regex and XPath which can be compared to each other. For fairness, we make sure that the queries are actually comparable. For example, we check that each of these queries are testing the same result set. Moreover, we verify that both Regex and the equivalent XPath return the same result set. The comparable

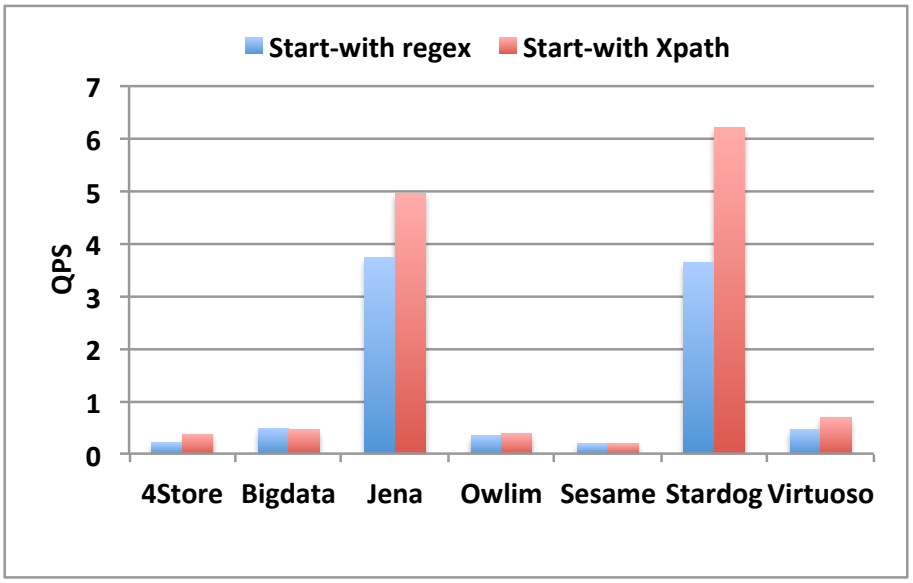


Figure 4.10: Starts-with query comparison between Regex and XPath.

queries between Regex and XPath are related to the starts-with, end-with and contains queries.

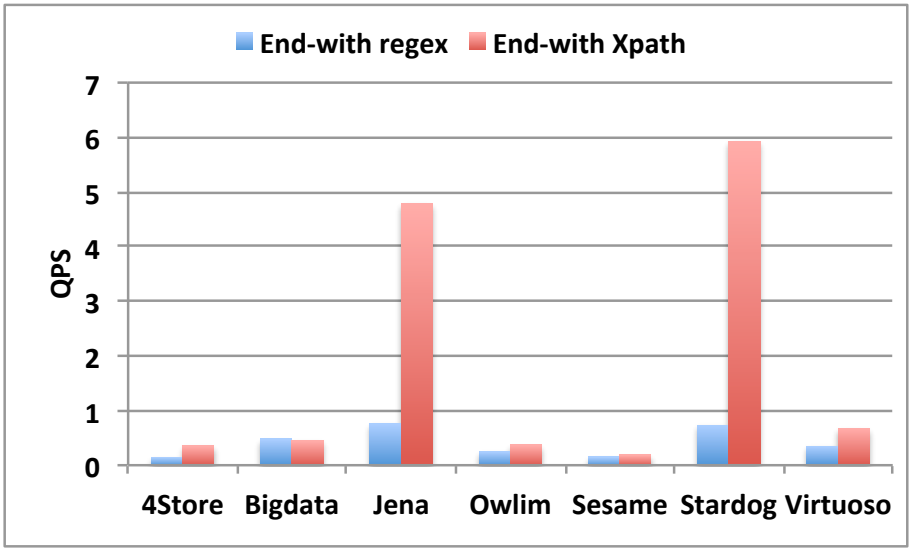


Figure 4.11: Ends-with query comparison between Regex and XPath.

Figure 4.10 illustrates the START-WITH method, using both Regex and XPath. Stardog and Jena show the same performance in Regex queries, but Stardog improves by almost doubling the QPS. Others indicate low performances in both Regex and XPath, while some have identical performance in both Regex and XPath, such as Sesame and Bigdata. Figure 4.11 is intended to test the END-WITH matching, which also shows a good performance for XPath, and is identical to their performances with the START-WITH matching. Stardog and Jena improve by around a factor of 5. Unlike XPath

however, Regex performs inefficiently. In the case of Regex, matching the START-WITH is more efficient (compared to other Regex features), where the engine would immediately start checking the start of the string and the pattern would then report a failure more quickly. In the case of END-WITH matching, the Regex engine, unlike both XPath and START-WITH within Regex, would have to traverse the characters to verify the end-with matching. XPath constantly finds the last character, and then it would only perform the matches. XPath would also do an end-with method in the same fashion for its start-with.

Figure 4.12 shows the CONTAINS method for both Regex and XPath. Note that it is not comparable to the other functions (START-WITH and END-WITH), since it tests a different predicate. CONTAINS query is applied on `rdfs:label`, which has lower records and therefore can be executed faster, compare the Y axis in Figures 4.10, 4.11 and 4.12. Stardog, Jena and Virtuoso perform better respectively. Jena has improved from Regex to XPath by a factor of 8, followed by Stardog with an improvement of a factor of 2.5.

Sesame and Bigdata show almost an identical performance for all the three functions (START-WITH, END-WITH and CONTAINS) in both Regex and XPath. There is a possibility that these stores are actually using a Regex engine for the XPath functions, that is by converting XPath patterns to an equivalent Regex one.

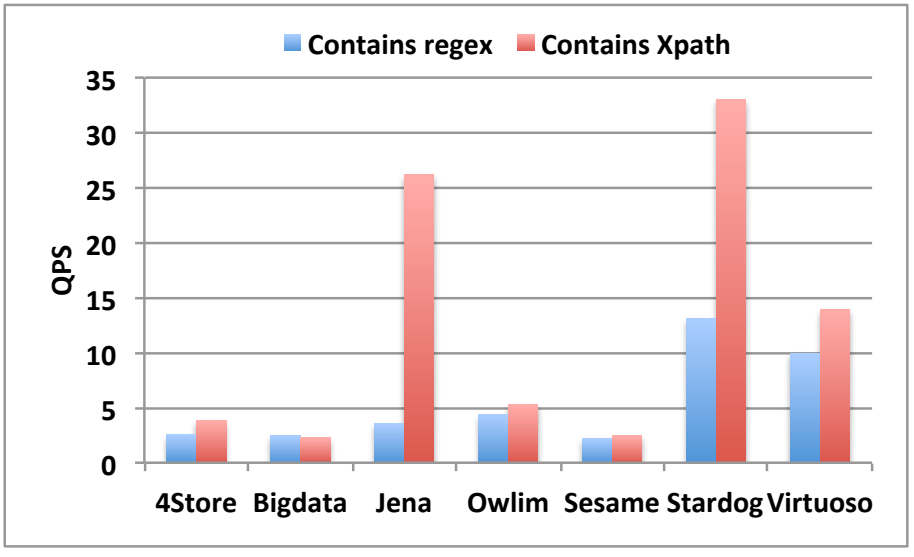


Figure 4.12: Contains query comparison between Regex and XPath.(Note: Y axis is different from the above figures).

4.2.7 The Original BSBM Against BSBMstr

It is possible to compare the general differences between benchmarks as long as they use the same metrics, such as QPS and QMPH. We, here, perform a comparison between

the original BSBM (Bizer and Schultz, 2009) and the Regex-specific BSBMstr. The original BSBM has 11 queries that cover a wide range of SPARQL features, such as different joins with $UNION$ and $OPTIONAL$. BSBM used to have a query to test the Regex filter; it, however, has been removed from the benchmark, due to its slow run in comparison to other queries, which makes that query dominate the overall results [13].

The original BSBM is run on the same machine as BSBMstr using the same configurations. This is done to have a comparable statistics between the two benchmarks using the same metrics. Figure 4.13 shows the result of running the original BSBM queries against the default chosen RDF stores by the original BSBM (Bigdata, Jena TDB, BigOwlIm, Virtuoso), using a dataset of 8 million triples posed by a single client. These four RDF stores which have been chosen by the original BSBM are all included within our benchmark BSBMstr beside some others. They are considered to be the top RDF stores by many other people such as (Guo et al., 2005; Duan et al., 2011; Saleem et al., 2015). The RDF stores have the same configurations that we have applied to BSBMstr.

Given the QMPH metrics, which refers to Query Mix Per Hour, we compare this figure in relation to Figure 4.2 in their general performance. The main finding is that the original BSBM run performance is in the order of 10,000 QMPH. On the other hand, BSBMstr runs in the order of 100 QMPH. This shows that BSBMstr queries have a slower performance than the original BSBM by two orders of magnitude. It should be noted that BSBMstr queries are simple in their structure, mainly consisting of one triple with a Regex filter. In the original BSBM, queries sometimes have multiple joins, filters and various features. This mainly shows the computational complexity of the Regex filter in relation to other features within SPARQL.

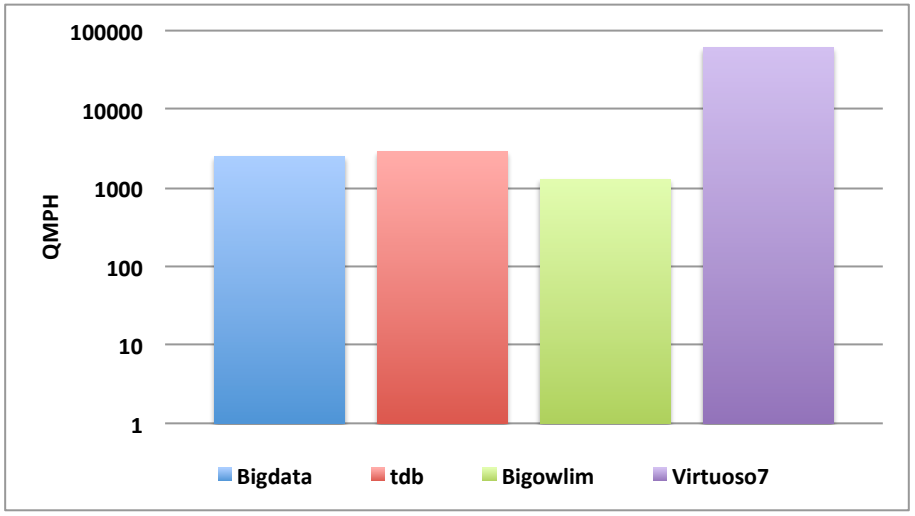


Figure 4.13: RDF stores comparison using QMPH against a set of the original BSBM queries using 8 Million triples of BSBM dataset.

[13] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/

4.3 Lessons Learnt

The benchmark helped to observe what affects the performance of Regex within SPARQL. We can list these factors, as follow:

- The average length of the tested literal. Therefore, those predicates which indicate that the object literal has a long length should be carefully executed with Regex.
- The number of the tested literal records. This causes expensive disk accesses.
- The complexity of the Regex pattern itself. In some cases, such as START-WITH Regex pattern, the length of the string literal is neglected, unlike END-WITH, which still requires scanning through the string literal until the last character, before being matched.

4.4 Conclusion

Regular expressions are the main facility for searching through strings within SPARQL specifications. However, this has not been adequately tested with current benchmarks. The original BSBM used to have one query to test Regex; it was then removed because it is very slow and would dominate the overall running time over other queries. In this study, we extended the Berlin SPARQL Benchmark through our proposal (BSBMstr) to include a set of Regex and XPath queries. Our experiment shows that Regex queries are affected by two main factors: disk accesses and Regex complexity. It also gave us some insight into how the length and number of tested records (cardinality) affect the Regex queries. BSBM, including BSBMstr are synthetic benchmarks, they have lately been criticised. In the next chapter, we revisit the issue of generic SPARQL benchmarking.

Chapter 5

CBSBench: Cluster-Based SPARQL Benchmark for Assessing the Performance of RDF Stores

SPARQL benchmarking has been a relevant topic almost since the start of the popularisation of Semantic Web technologies, such as SPARQL and RDF. An example of the main benchmarks in a chronological order is: the Lehigh University Benchmark (LUBM) (Guo et al., 2005), the SPARQL Performance Benchmark (SP^{2}Bench) (Schmidt et al., 2008), the Berlin SPARQL Benchmark (BSBM) (Bizer and Schultz, 2009), the DBpedia SPARQL Benchmark (DBSB) (Morsey et al., 2011), the Waterloo SPARQL Diversity Test Suite (WatDiv) (Aluç et al., 2014), and the Feature-Based SPARQL Benchmark (FEASIBLE) (Saleem et al., 2015). SPARQL benchmarking also has attracted some projects, such as the Linked Data Benchmark Council (LDBC) (Angles et al., 2014), which is a European Union (EU) project that aims to provide large scale benchmarks for graph and RDF BDMSs. Holistic Benchmarking of Big Linked Data (HOBBIT) (Ngonga Ngomo and Röder, 2016) is another project with similar goals to provide better adoptions of linked data. There have also been workshops that have been specialised in the area of SPARQL benchmarking, such as the last 2nd international workshop on Benchmarking RDF Systems (BeRSys 2014), in conjunction with the 40th International Conference on Very Large Data Bases (VLDB2014)[1], and Querying the Web of Data workshop (QuWeDa 2017), in conjunction with the Extended Semantic Web Conference (ESWC2017)[2].

[1] http://events.sti2.at/bersys2014/

[2] https://sites.google.com/site/quweda2017/

There are two major issues with the current benchmarks: firstly, most of them are based on synthetic datasets and/or queries. This can produce biased results as they may not represent real-world case scenarios. The second issue is that almost all current benchmarks (including synthetic and real-world benchmarks) are based on a fixed set of queries. Even though some benchmarks implement what is sometimes referred to as templates, they still have generally fixed structures. In both cases, having to run benchmarks on a fixed set of queries may cause RDF stores to use caching mechanisms, which may not accurately reflect their performance, assuming that benchmarks run their queries multiple times to get to a steady performance. In addition to this, RDF stores may be tuned to optimise their stores on these fixed set of queries.

It might be unavoidable to use query templates within benchmarks that are based on synthetic queries, such as BSBM or WatDiv, as it would be impractical to write many synthetic queries. For the other type of benchmarks which use real queries (such as DBSB or FEASIBLE), there is usually a large set of queries. These benchmarks usually aim to reduce queries to a representative set of queries to run the benchmark. In the case of DBSB, parameterised queries out of each clusters. The reduction of a large amount of log queries is not only wasteful of valuable data, but also may not be representative. In this chapter, we aim to answer the research question (3):

What are the issues with the current generic SPARQL benchmarks?

This chapter has led to three main contributions:

- It proposes a new method for clustering SPARQL log queries based on their structural features using K-means++.
- It also proposes a new benchmark methodology which unlike other benchmarks uses as many queries as possible. The result of the benchmark is not to test the performance of different queries but, rather, the performance of different clusters of queries.
- It reports the results for five known RDF stores that have been tested against our benchmark.

5.1 Related Work

There are a good number of SPARQL benchmarks in the literature. LUBM, as one of the first benchmarks, tests the OWL capabilities in RDF stores. It relies on a synthetic dataset and queries that describe academic institutions and their universities, departments, professors, students and so on. The structure of the dataset was somehow hierarchical; the more universities exist, the more departments and professors there can be. SP^2Bench is also an early benchmark that has a synthetic dataset and queries. The

dataset is built around Digital Bibliography and Library Project. It has more SPARQL features than LUBM, such as $FILTER$ and $OPTIONAL$. BSBM has a number of use cases, such as explore, business intelligent and update. BSBM introduces more realistic runs that mimic a real users' operations, such as multiple clients sending queries at the same time. The main issue with these benchmarks has been deeply analysed by Duan et al. (2011). Their paper concluded that these benchmarks do not accurately resample a realistic scenario of RDF data, by having a structure of RDF that is rather similar to a relational one.

The second set of benchmarks are more related to this current work. Instead of generating synthetic datasets, perhaps it is more realistic to reuse existing ones. DBSB is the first benchmark for SPARQL that uses a real dataset (DBpedia), as well as their query log files, as their set of queries. To cluster the log queries, a Levenshtein similarity is calculated on the query string. The problem with this approach is that differences between queries would be affected by URIs or literals, rather than the general structure of the queries. An updated version of this benchmark (Morsey et al., 2012) includes a set of vectors/features to be clustered with Border-flow (Ngomo and Schumacher, 2009). Some of these features have minor performance effects in comparison to others. For example, $LANG$, STR or $isLiteral$ and other similar features are being removed from the feature set as performed in FEASIBLE (Saleem et al., 2015).

FEASIBLE is probably the most recent and closest related work. We both use real datasets and their log queries. They, however, consider both the structural and data-driven features, described by Aluç et al. (2014). In our case, we mitigate the effect of data-driven influence by applying some restrictions on the queries such as $OFFSET$ and $LIMIT$. By doing so, reasoning over the various effects of SPARQL's structural features becomes more clearer. FEASIBLE also selects a prototypical query from each cluster, an approach adopted from (Morsey et al., 2012). There are also differences in the clustering methods used. FEASIBLE implements the LIME framework, which we see as a k-medoids method (although the paper did not mention it by that name). That is, by performing the clusters based on the data points. In this paper, clustering is done by using k-means, which takes centroids into consideration rather than the points themselves.

In our work, it is intended to use as many queries as possible within each cluster. Unlike all previous benchmarks, we present our main results as the performance of different clusters, each of which have a set of features, rather than queries. For example, instead of presenting Query x, as slow because it has $OPTIONAL$, we say cluster x is slow because the cluster has a high usage of queries that have $OPTIONAL$.

5.2 Preliminaries

The two main concepts of this work are related to both the RDF data model and its query language SPARQL. Firstly, we define the SPARQL language. These definitions are not meant to be exclusive, only features that are of interest to this study.

Definition 5.1 (SPARQL Query)**.** We assume a SPARQL Query $sp = (QF, BGP, BGR_mf, Mf)$, where Query Form QF $\in (SELECT \cup CONSTRUCT \cup DESCRIBE \cup ASK)$ and disjoint. As well as BGPs modifiers $BGR_{mf} \in (UNION \times OPTIONAL \times FILTER \times REGEX)$. Finally, general Query modifiers $MF \in (LIMIT \times OFFSET \times ORDERBY \times DISTINCT)$.

Definition 5.2 (Basic Graph Patterns BGPs)**.** BGPs are sets of triple patterns $|T|$. A triple pattern $t(s,p,o) \in (U \cup V) \times (U \cup V) \times (L \cup U \cup V)$, where U, V and L are URIs, Variables and Literals respectively. Moreover, $L \cup U \subseteq C$, where C is a constant. These different combinations result in various graph shapes and joins.

Definition 5.3 (BGP Joins Representations)**.** We also assume Star Shape SS for the whole query $SS_{qf} = \Sigma\big(t(s) = \bar{t}(s) \cap t(o) \neq \bar{t}(o)\big)$. Moreover, Chain Shape CS is also another shape where $CS_{qf} = \Sigma\big(t(s,p,o) = \bar{t}(o,\bar{p},\bar{o})\big)$. To utilise joins, we assume two representations that involve permutations of different types:

- Position-Type Joins: $PTJ_{qf} \in (Sub.Sub \times Pre.Pre \times Obj.Obj \times Sub.Pre \times Sub.Obj \times Pre.Obj)$, where $Sub.Sub \in t(s \bowtie \bar{s}) \cup t(\bar{s} \bowtie s)$ and so on for the rest.
- Resource-Type Joins: $RTJ_{qf} \in (VVC \times VCC \times CVV \times CVC \times VCV \times CCV)$, where $VVC \in t(s,p,o) \in V \times V \times (L \cup U)$ and so on for the rest.

These definitions are mainly introduced to explain the process of extracting the syntactic features of SPARQL queries to be used within the clustering method. An example of all these definitions is given in Section 5.4.2 using a SPARQL query as an example in Listing 5.1.

5.3 Data Generation (Workloads)

Most of the proposed benchmarks have been designed based on synthetic datasets such as BSBM, LUBM or SP^{2}Bench. Producing different workloads is not a problem in this case as they are usually based on modifying a scale factor. The main issue is that the graph of data would increase/decrease while maintaining the same "structureness"; these approaches, therefore, treat graph data as though it were relational data. Duan et al. (2011) discusses the issue that most benchmarks tend to assume a fixed structure, which is not the case with real graph data.

On the other hand, it is rather challenging to produce a representative dataset from a real one. The naive approach would be to conduct a random-based selection based on a scale factor that is relative to the size of the dataset. Instance-based selection has been proposed by Morsey et al. (2012); randomly selecting instances and retrieving all sub-triples while holding the condition that the overall produced triples must not go beyond the scale factor relative to the size of the dataset. Duan et al. (2011) proposed the Coverage and Coherence model. The focus of this model is more about the structureness of the graph rather than about its instances. For example, given a 1 for the coherence parameter would mean a high structureness and, therefore, all instances should have the same properties.

In this work, it is useful to choose the method that would produce the highest result sets. The more results being retrieved, the slower a query will perform and, therefore, differences between RDF stores become more noticeable. We implemented the Instance-based approach for producing different workloads at 50%, then 10%, of the original dataset size.

5.4 Query Processing

The goal at this stage is to provide a set of clusters, each of which has a set of queries that are similar in their query structure. In the same way as having the dataset generated from a real-world dataset, the queries are also processed from log files that have been posed on the DBpedia SPARQL endpoint. Figure 5.1 illustrates the general work flow of processing queries which were made ready for running the benchmark.

5.4.1 Query Selection and Filtering

For both the dataset and the query log file, we applied our experiment on version 5.3.1 of DBpedia which has a total of 3,182,389 raw queries. Although there are other newer versions, we found this version to be suitable for the machine we applied our experiments on in relation to the size of the different workloads, especially when handling the 100% workload. Moreover, we wanted to be consistent with other benchmarks which have used the same version, such as (Morsey et al., 2012; Saleem et al., 2015). For this step, we follow the same steps as in (Morsey et al., 2011), which apply Query Variations, but we kept all queries even with fewer frequencies. Unlike Morsey et al. (2012) in which there is one query to be nominated for each cluster, we aim to keep all queries, as they are all going to contribute in running the benchmark.

The structural features of SPARQL are not the only factors that influence the performance of query execution, Data-driven properties also have an effect (Aluç et al., 2014;

Saleem et al., 2015). For example, two queries with the same structure can have different performances if they have differences in the number of intermediate results to be processed. Nevertheless, both sets of properties (structural and data-driven) would have a major effect on the clustering process. Unlike (Saleem et al., 2015), where the two properties have been taken into consideration, we aim to restrict this study to the structural properties for a more deeper analysis. To alleviate the data-driven effect, first, all queries that do not yield any results are omitted. Then, a LIMIT operator with up to 10 results is added to all queries.

There are still issues with complying to SPARQL standardisation (Bizer and Schultz, 2009). Different RDF stores may have their own extensions, such as Full-text search (FTS) as a approximate matching, geo-spatial or analytical extensions. In this case, DBpedia query log files are produced by Virtuoso, which also has its own extensions. To maintain a consistent test, we removed all queries with such extensions.

The overall number of queries to be clustered and then evaluated using this benchmark is 1,280,930 queries. To check the distribution of these queries against each cluster, see the x-axis in Figure 5.4. The figure shows each cluster and its percentage of queries against the overall number of queries. For example, cluster 1 has 23% of queries which means it has around 294,613 queries.

```
SELECT DISTINCT *
WHERE
  { ?var1 rdf:type dbpedia-owl:Person .
    ?var1 rdfs:label "person"@en
    OPTIONAL
      { ?var1 dbpedia-owl:nationality ?var3 .
        ?var3 rdfs:label ?var0
        FILTER regex(?var0, "keyword")
      }
  }
OFFSET 0 LIMIT 10
```

Listing 5.1: A SPARQL query example.

5.4.2 Feature Matrix Generation

For each query, a set of vectors is generated, corresponding to various SPARQL features. This benchmark is designed for testing the performance differences between various RDF stores. Therefore, it is not intended to provide an inclusive set of features. In addition to this, the more features we add, the more sparse the clusters would become. To the contrary, it is intended to focus on the most performance-sensitive features across SPARQL.

As an example, we give the SPARQL query in Listing 5.1 and go over the different SPARQL features. We start with Query Form QF, which is a binary vector as well as General Query modifiers MF, whereas BGPs modifiers BGR_{mf} are numeric vectors (0,1,2,...,n), equal to how many times they appear within the query. For the SPARQL example, $SELECT$, $DISTINCT$, $FILTER$, $OPTIONAL$ and $REGEX$ will have a value of 1 or 0 for the rest of this class. $OPTIONAL$ and $UNION$ clauses are related to joins and are computationally expensive. One study has suggested that $OPTIONAL$ falls in the category of PSPACE-complete (Pérez et al., 2009). We do not include $OFFSET$ and $LIMIT$ as they are enforced on all queries to restrict their result set. This restriction is applied to reduce the side effect of data-driven influence on query performance. Data-driven influence is also a major part of how a query can perform, such as cardinality and result size (Saleem et al., 2015). Restricting the data-driven influence has been done using a limit and offset on each query, which can help in reducing the influence of these data-driven features. This study, then, can focus on the syntactic features and how they affect the performance of SPARQL queries.

One of the features that has been most discussed in the literature is the complexity of joins (Saleem et al., 2015; Aluç et al., 2014). To measure joins across different queries, we adopted the approach presented by Arias et al. (2011), which is aimed to describe the state of SPARQL queries posed by real users. To assess joins within each query, we adopted three main measures (described in Definitions 5.2 and 5.3). We calculate the total number of triple patterns defined in Definition 5.2 which results in the example as $|T| = 4$. Definition 5.3 describes the different types of joins between triple patterns. Given the SPARQL query and the first type Position-Type Joins, PTJ_{qf}: $Sub.Sub = (2)$ $Sub.Obj = (1)$, and 0 for the rest. Then, we construct eight permutations of whether a resource is variable or constant denoted as Resource-Type Joins RTJ_{qf}: $VCC = (2)$ $VCV = (2)$ and the rest with 0. Finally, we measure the degree of the two main SPARQL shapes. For Star Shape SS_{qf}, we calculate the out-degree of each resource, whereas for the Chain Shape CS_{qf} we look at the connectivity of an object to be a subject across triple patterns. For the example query, $SS_{qf} = 2$ and $CS_{qf} = 1$. Table 5.1 also shows each feature along with their values resulting from extracting features on the SPARQL language example 5.1. This results in 17 features for measuring joins within each query and 10 other general features. This process is applied to all 1,280,930 queries.

5.4.3 Query Clustering

Before running the clustering method, we normalise all numeric vectors to a scale between (0,1) and relative to the range values of that vector (vector-wise), using $f(x) = x - min(x)/max(x) - min(x)$.

The process of clustering is important for grouping queries with similar sets of features. Given data points of $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, where $\mathcal{X} \subset \mathbb{R}^d$, as well as a pre-defined number

feature(b)	value	feature(n)	value	feature(n)	value	feature(n)	value
Select	1	Optional	1	ObjObj	0	VCV	2
Construct	0	Union	0	SubPre	0	CCV	0
Describe	0	Filter	1	SubObj	1	Triple patterns	4
Ask	0	Regex	1	VVC	0	StarShape	2
Order by	0	SubSub	2	VCC	2	ChainShape	1
Distinct	1	PrePre	0	CVV	0		

Table 5.1: Result of feature matrix generation process on SPARQL query 5.1. Note: (b) refers to boolean features which their values are either 0 or 1, and (n) refers to numerical features which their values accept numerical numbers (0,1,2,3 ...,n).

of clusters as K, each of which has a centre as C, such that $K \leq |\mathcal{X}|$, assign each $x \in \mathcal{X}$ into K, given a minimum distance of $d(x, \bar{x})$ within C, the result would be that $k \in K \subseteq \mathcal{X}$.

In our approach, we apply the unsupervised and well-known K-means algorithm based on numerical attributes/features which uses Euclidean Distance for measuring the centroids. To avoid the computational problem (NP-hard) of running the standard K-means method, we adopt K-means++ (Arthur and Vassilvitskii, 2007), which implements a seed-based randomisation technique for the initial centroids positions and runs in $\Theta(log\ k)$-competitive.

Since the clustering algorithm requires a pre-defined number of clusters, it is important to choose a suitable number. The role of thump is that, the more clusters are used, the easier it becomes to reason about the performance of each cluster. This is because there would be a smaller set of features contained in each cluster. This role is more relevant to the Query Per Second (QPS) metric, since it tests the performance of each cluster given alongside its set of features. Query Mix Per Hour (QMPH), on the other hand, may be less affected by the number of clusters, as it accumulates the overall performance regardless of the number of clusters. In this experiment, we chose 25 clusters to conform with the previous benchmark, DBSB (Morsey et al., 2012). Each cluster also has the ratio of each feature existing within that cluster (see Figure 5.4).

Here, we show how to use Figure 5.4 to reason over the relation between clusters and features. For example, the figure shows that cluster 23 have around 2% of the overall queries, which is 25,618 queries. The figure also shows that there are around 100% of $DISTINCT$ that exist within that cluster. $DISTINCT$ is a binary feature which cannot be presented more than once within each query and, therefore, is denoted with 1 as the maximum quantity. This means that all queries within the cluster actually have this feature. The same cluster also have around 40% of Sub-Obj joins. Since the maximum Sub-Obj is 4, we can conclude that there would be around 2 Sub-Obj within each query within cluster 23.

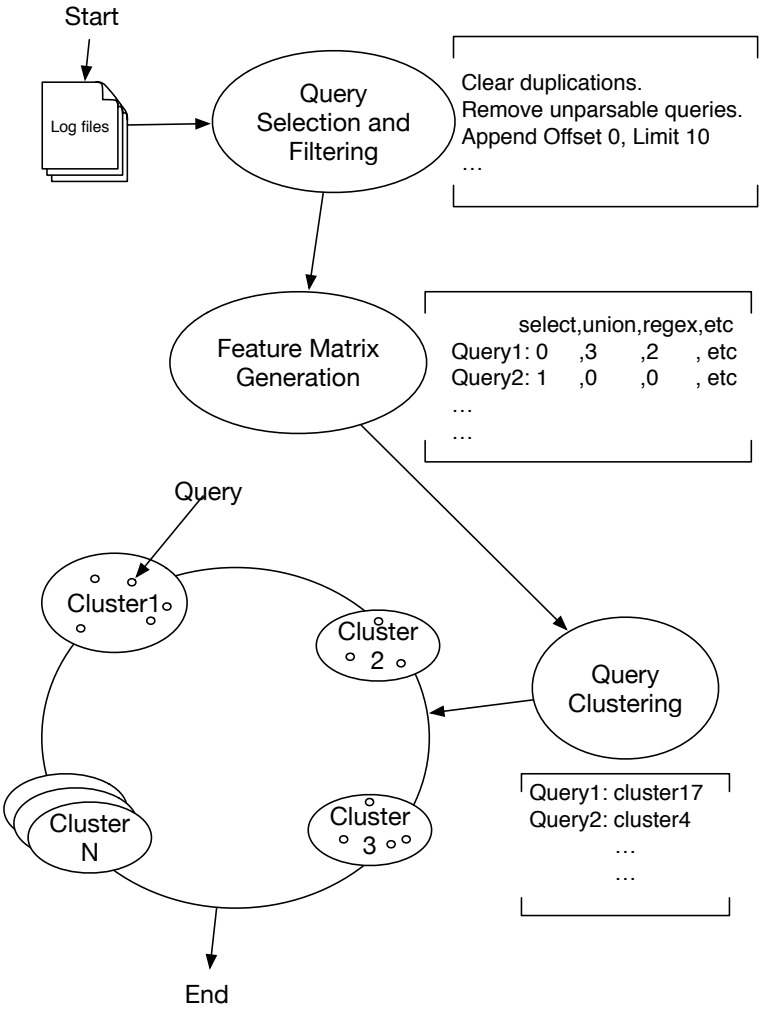


Figure 5.1: CBSBnech queries processing work flow.

5.5 Benchmark Design

Most of the previous benchmarks such as LUBM, BSBM and DBSB implement parameters within queries to trick the caching mechanism. They are, however, still prototypical queries with a static query structure. Even within DBSB, it nominates a query from each cluster to be the prototypical one across the cluster. Within FEASIBLE, no parameters are implemented; instead it produces a fixed larger number of queries, such as 125. We argue that reducing queries within a cluster is not the optimal way, and that even having to implement parameters across the queries would still maintain the structure of the query. Given that there is a rich range of queries within each cluster, it is perhaps, reasonable to use as many of them as possible. Figure 5.2 shows the general design of how CBSBench runs queries against a given RDF store. Algorithm 1 also depicts the main algorithm and metrics for this benchmark. As in BSBM, DBSB and FEASIBLE, we also implement the two benchmark metrics: QPS and QMPH. Within every run, each of the 25 clusters will be visited once. During each cluster visit, we

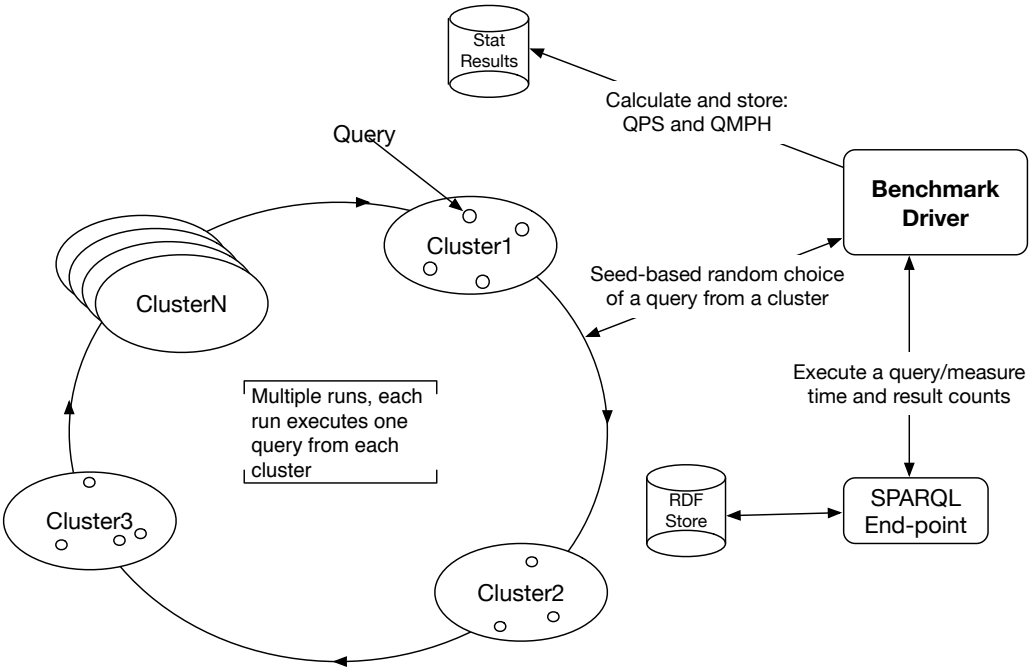


Figure 5.2: CBSBench overall running design.

apply a seed-based random function to determine which query within the cluster would be executed. The variation of queries, even within the same cluster, applies more stress on the RDF store. Moreover, the seed-based random function ensures that the same query sequence are going to be maintained across different RDF stores.

Algorithm 1: Benchmark Core Algorithm

Data: Query clusters c, number of runs $runs$.
Result: QMPH and QPS.
Initialisation;
Warm up runs;
foreach *Run* **do**
  foreach *Cluster c* **do**
    $Rand \leftarrow$ generate seed-based random number $\leq |c|$
    CAETime $\leftarrow$ accumulate Cluster Average Execution Time of query($Rand$)
  end
  RAETime $\leftarrow$ accumulate Run Average Execution Time
end
QPS of each cluster $\leftarrow$ inverse function of CAETime
QMPH $\leftarrow$ inverse function of RAETime

5.6 Evaluation and Results

In this section, we run two different types of evaluations. We published the benchmark driver, query clusters, features generator and other raw data on GitHub[3]. Firstly, we evaluate whether our clustering method (K-means) alongside its feature set produces good clusters. In this context, we refer to a good clustering method if the execution time of a particular cluster has less performance variation than other approaches. The second part of the evaluation is done by applying the benchmark on different RDF stores.

Appendix C presents details from various technical points, such as the clustering algorithm with Weka[4], the resulted clusters with their queries and the raw results.

5.6.1 RDF Stores and Configurations

There has been a large amount of work on building high performance RDF stores. Some of the early proposals include 3Store (Harris and Gibbins, 2003), Sesame (Broekstra et al., 2002) and Jena (Wilkinson et al., 2003). All these RDF stores were either completely, or had a version that is built on top of a relational data model. These stores have shown slow performance rate, perhaps due to the high number of generated joins as SPARQL is translated into SQL (Tsialiamanis et al., 2012). RDF-3x (Neumann and Weikum, 2008) was a novel approach to shifting the back-end design to a native model. Since RDF statements consist of three resources (Subject - Predicate - Object), a six permutations indices seems to be working well. Sesame, Jena and many others have adopted this model into their stores.

We benchmark as many RDF stores as possible as long they are actively supported and expose a SPARQL endpoint. The area of RDF stores is still new compared to mature BDMSs systems such as relational databases. Therefore, there are some RDF stores that have not been included, because they do not support some SPARQL features, such as $FILTER$ as in RDF3x, gStore. The candidates for this benchmark are as follows: GraphDB (formerly as OwlIM-SE), Jena TDB, Sesame native, Virtuoso7, Bigdata, 4Store and Stardog. However, 4Store and Stardog have been later excluded because of their poor performance, in that they failed to produce any results for the 100% workload.

The dataset that is used in this evaluation is the English version of DBpedia (3.5.1)[5]. It is also chosen to conform with previous benchmarks such as FEASIBLE and DBSB.

[3] https://github.com/SaudAljaloud/SPARQLlogClusters

[4] http://www.cs.waikato.ac.nz/ml/weka/

[5] http://wiki.dbpedia.org/services-resources/datasets/data-set-35/data-set-351

This dataset has around 200 million triples, which takes around 37 Gigabytes of plain N-triples file format[6]).

RDF stores are usually not plug and play; rather they need some degree of set up configuration. To ensure that no configuration bias was introduced, we followed the approach of BSBM. RDF store vendors had previously been contacted about the initial set up of their store. Each vendor was encouraged to give their feedback and comments on the general set-ups. Not all vendors replied, such as Bigdata. For this, we took no further actions and simply followed their documentation as closely as possible. Virtuoso, GraphDB, Sesame and Jena gave more feedback which was taken into consideration. Their comments were mainly around memory allocation. For example, Jena (with Fuseki) implements off-heap memory usage, which does not require allocation from within the Java heap space. As different stores have various implementations, we allow RDF stores to use all machine commodities (memory and processors).

For each workload (10, 50 and 100 respectively), we follow the same procedure for all runs against all RDF stores. Firstly, we load the dataset. Then, the RDF store is stopped and the RAM is cleaned to avoid any caching applied by the operating system. Then, we again run the RDF store and run the benchmark. These steps are taken from the BSBM benchmark driver rules.

5.6.1.1 Experiment Set-up and Hardware

The benchmark is built on top of the BSBM[7] benchmark driver rules. It is built on Java and execute the SPARQL queries through HTTP requests to the targeted RDF store. For each RDF store, we (1) load the dataset, (2) shut-down the RDF store, (3) clear the Operating System caches, (4) run the RDF store again. Then, we run the benchmark.

We run the benchmark on four different sets of client numbers (1, 4, 8, 64), as implemented by BSBM. For each of them, we start a warm up run (5, 8, 16 and 128) with respect to clients. For example, before executing the actual run for a single client (client 1), we do a warm up run for 5 times. During running 4 clients, we do 8 warm up runs, and so on. Once the warm up has finished for each run, an actual run is started with (50, 32, 64, 128) number of runs, respectively. The number of actual runs is also implemented by BSBM with the same parameters. This is because not all RDF stores have managed to keep up running in the main single client run (especially for Bigdata). For each run, each of the clusters will be visited according to the number of clients. The whole process is also repeated for each workload (100%, 50% and 10%). We also restrict the time-out limit of each workload (360s, 180s and 36s) with respect to the workload. For example, all 100% workloads are given 360 second as time out, and so on for other workloads.

[6] http://downloads.dbpedia.org/3.5.1/en/

[7] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BenchmarkRules/index.html#rules

Triple Store	Sum of Coefficient Variation *CV*		
	FEASIBLE	Our proposal	Improvement
Sesame	65	36.7	56.5%
Jena	55.5	47.8	13.9%
Virtuoso	51.6	43	16.7%

Table 5.2: SE on different RDF stores against two clustering approaches.

Any query which reaches that limit would be assigned the limit of that workload, which is the maximum execution time.

All experiments have been carried on a dedicated Linux server that has: 2x AMD Processor 2.8GHz and 8 cores, DDR3 128GB RAM, 1066MHz 2x 300GB of rotational disks.

5.6.2 Sum of Coefficient Variation

We first evaluate the clustering approach compared to FEASIBLE. The comparison involves both the clustering approach and the quality of the feature sets combined. As the two approaches are not identical in their procedures and the shape of final results, we modified FEASIBLE to work on our model. First, we set K to be 25, then run FEASIBLE to produce the set of partitions in the form of: $\mathcal{X} = \{\{q1, q2, q3, q5\}, \{q4\}, ...,$ where $x \in \mathcal{X}$ is equivalent to $c \in C$.

Then, we removed queries where full scanning has taken place, as well as applying $OFFSET$ to be zero and $LIMIT$ with 10 for all queries. At this point, we could compare the set of two clusters in their performance variation against our benchmark. To evaluate each approach, we calculate the sum of errors or the Coefficient Variation, denoted as CV. For each cluster $c \in C$, we calculate the mean σ_c of all execution time of that c, then divide it by its standard deviation μ_c. Finally, we sum up all the result for all clusters which indicate the sum of the Coefficient Variation $CV = \sum_{c \in C} \frac{\sigma_c}{\mu_c}$. Less CV indicates fewer errors within the set of features applied as well as the clustering approach. We apply this method on three different RDF stores. Table 5.2 shows that our approach has fewer CV than FEASIBLE within the chosen three RDF stores. This means that our clustering method and the set of features produces clusters, each of which has queries with execution times closer to each other.

5.6.3 Benchmark Result on RDF stores

To evaluate the different RDF stores, we rely on two main metrics that have been used in various benchmarks: QPS and QMPH. These two metrics are inverse functions of the average execution time of each one. We use QPS (fine-grained) to evaluate the different

RDF stores against different clusters when run as single clients on 100% of the dataset used (DBpedia). QMPH, however, is used for the coarse-grained analysis, by comparing the performance of RDF stores as a whole, regardless of the clusters. We also use it to compare RDF stores when smaller workloads of DBpedia are applied of 50% and 10% respectively. Finally, QMPH is used to compare the RDF stores performance when different clients are used to fire queries in parallel (single, 4, 8, 64) clients respectively. These tests are mainly taken from Bizer and Schultz (2009), although in our approach we consider clusters instead of queries.

5.6.3.1 Query Per Second (QPS) Analysis

To analyse the performance of different clusters over the RDF stores, we look at Figure 5.3 in relation to Figure 5.4. The overall performance of the different RDF stores is quite leaned around 10 QPS, with a few exceptions. This is mainly because of the $LIMIT$ being restricted to 10 for all queries. Since the size of the result set is a dominant factor, only those exceptions can tell us more about how different features affect the performance. For example, Jena's overall performance was exceptional, apart from cluster 2 which has mainly $CONSTRUCT$ queries. Those queries have been shown to perform badly within Jena by Saleem et al. (2015) . Clusters (6, 11 and 15) have queries about languages other than English, a feature that is not depicted by our feature set, also decreases the performance of Jena, since it causes a full scanning of triples with no results being returned. The worst performance was for Bigdata in cluster 23 with 0.01 QPS being reached, probably due to the high usage of various features, including $DISTINCT, UNION, FILTER$ and multiple joins.

A detailed look at the distribution of the execution time shows spikes of running time which may have been caused by the Java garbage collection affect. In clusters (21,6), Sesame has a notably poor performance, although the cluster contains simple queries (no joins for 21). The reason for this could be that the cluster mainly consists of triple patterns that are of type (VVC), which requires the OPS index that has not been constructed for Sesame (only the default SPO,POS are built). Such an observation cannot be picked out by other benchmarks, including FEASIBLE. Virtuoso, however, maintained a good performance in most of the clusters. GraphDB comes second, as it has a steady performance although in clusters (3,5 and 25), which they have CVV in common, seems to affect its performance. Probably, the simplest cluster is (19) which has no features and it only contains a single VCC. Jena outperforms with 62 QPS.

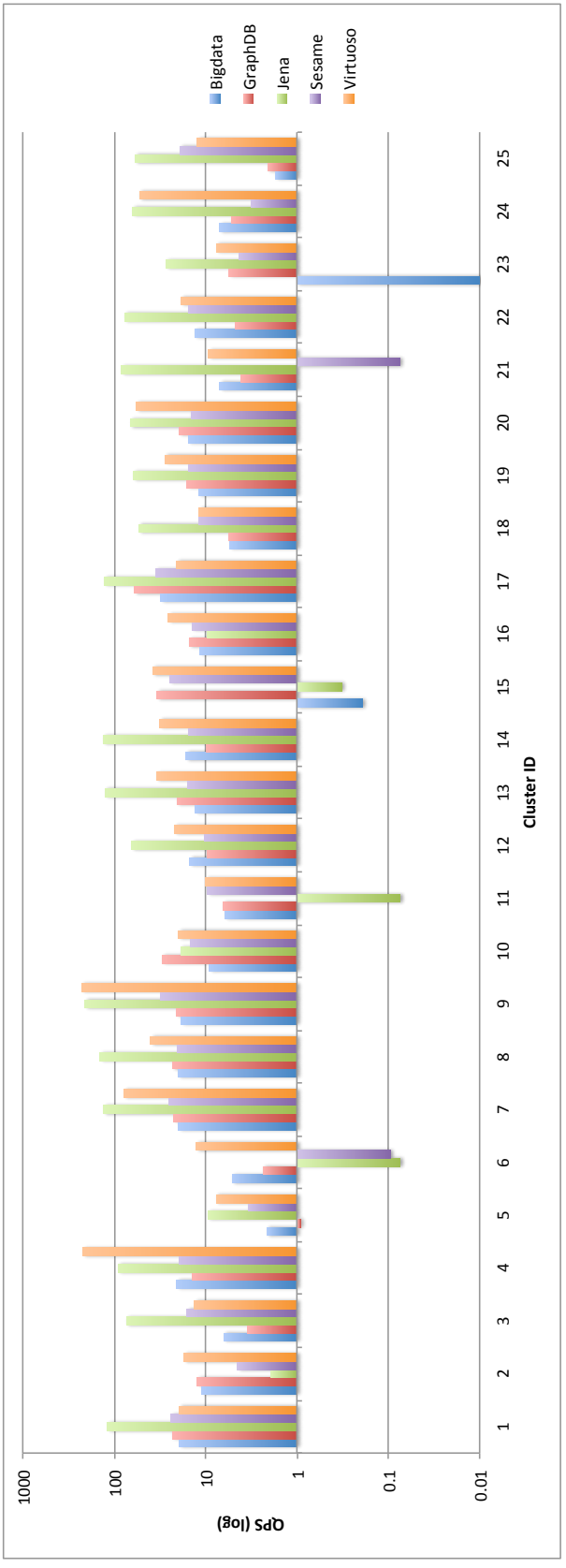


Figure 5.3: QPS for the different clusters of DBpedia queries using single client (the higher the better performance).

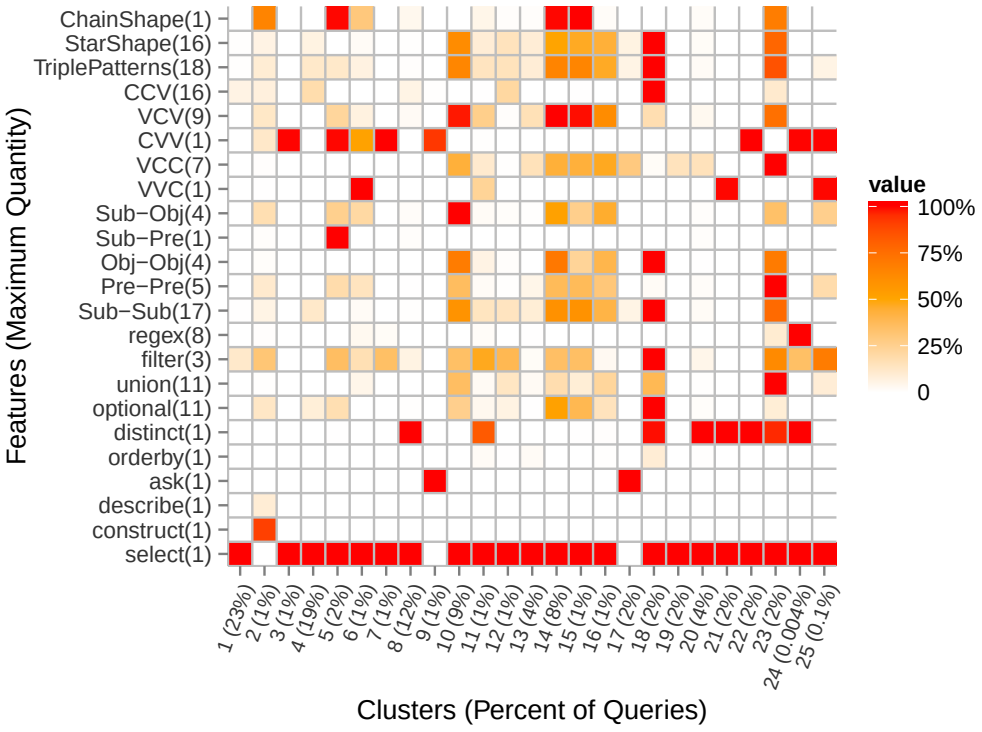


Figure 5.4: A heat map showing the existence of different feature ratios to their maximum within different clusters using DBpedia log files (Note that the minimum quantity of each feature is 0 except for TriplePatterns and ChainShape are equal to 1).

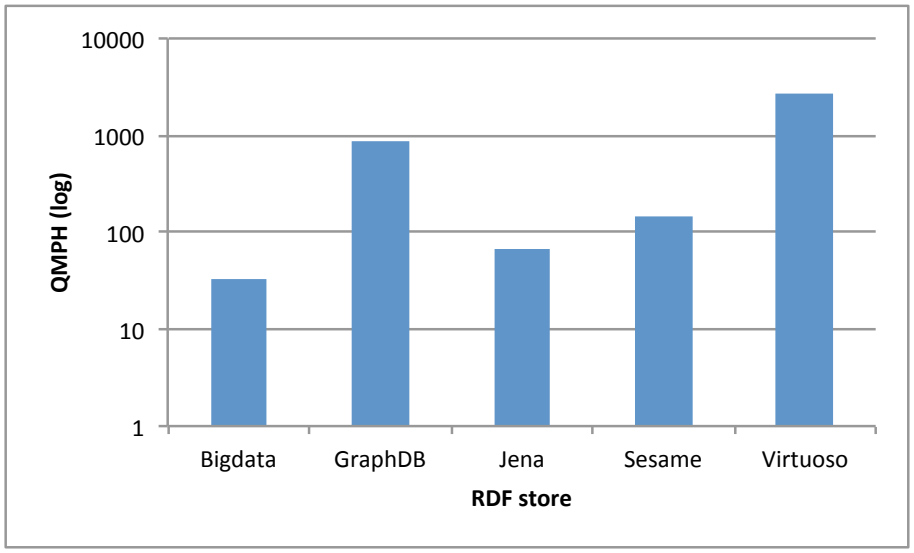


Figure 5.5: An overall comparison of RDF store performance using a full size DBpedia in QMPH (the higher the better).

Store	1	2	3	4	5	6	7	8	9	10	11	12	13
Bigdata	1.2	0	9.02	1.3	0	7.58	**0.46**	1.08	0	0	1.5	1	1.2
GraphDB	1.2	3.98	9.02	1.3	0	**7.6**	**0.72**	1.08	0	0	1.5	1	1.2
Jena	1.2	0	9.02	1.3	0	7.58	**0.48**	1.08	0	0	1.5	1	1.22
Sesame	1.2	3.98	9.02	1.3	0	**7.6**	**0.72**	1.08	0	0	1.5	1	1.2
Virtuoso	1.2	0	9.02	1.3	**9**	7.58	**0.46**	1.08	0	0	1.5	1	1.2

Store	14	15	16	17	18	19	20	21	22	23	24	25
Bigdata	0.18	0	1.9	0	3.1	2.4	1.02	4.8	4.78	**2.041**	6.4	10
GraphDB	0.18	0	1.9	0	3.1	2.4	1.2	4.8	4.78	2.16	6.4	10
Jena	0.18	0	1.9	0	**2.217**	2.4	1.2	4.8	4.78	2.16	6.4	10
Sesame	0.18	0	1.9	0	3.1	2.4	1.2	4.78	4.8	2.16	6.4	10
Virtuoso	0.18	0	1.9	0	**2.98**	2.4	1.2	4.8	4.78	2.16	6.4	10

Table 5.3: Result set average number on different RDF stores run on single client and 100% workload.

5.6.3.2 Result Sets Analysis

The speed of executing a SPARQL query is highly correlated to the number of results it returns. This benchmark runs a large variety of queries. One benefit is that the benchmark can detect implementation issues with RDF stores on certain SPARQL features. Table 5.3 lists the average result sets on each of the RDF stores based on their clusters. The table indicates that these RDF stores are generally consistent in their result sets. One possibility is that this is a result of the SPARQL standardisations (Prud'Hommeaux and Seaborne, 2008).

There are, however, still some minor differences between RDF stores. These differences could be caused by the query exceeding the time limit initially assigned (time-out). For example, in cluster 23 Bigdata has a lower result set than the others. This can be explained from Figure 5.3 which shows Bigdata to perform poorly. Clusters that have result sets equal to zero are mostly caused by other SPARQL query types such as *ASK* or *CONSTRUCT* as in clusters (2,9 and 17). A detailed analysis shows that other issues are the result of some queries having filters about languages other than English, as in clusters 5, 10 and 15. Cluster 7 shows differences due to queries that are URL encoded, which seems to be only correctly processed by GraphDB and Sesame.

5.6.3.3 Query Mix Per Hour (QMPH) Analysis

In this section, we report the results of running the benchmark from other perspectives, such as overall performance, different workloads and parallel executions (clients). Figure 5.5 indicates which RDF store can execute more queries per hour in a logarithmic scale. Virtuoso was able to execute more than 1,000 queries per hour, and was followed by GraphDB. Although Jena was performing quite fast in most of the clusters in Figure

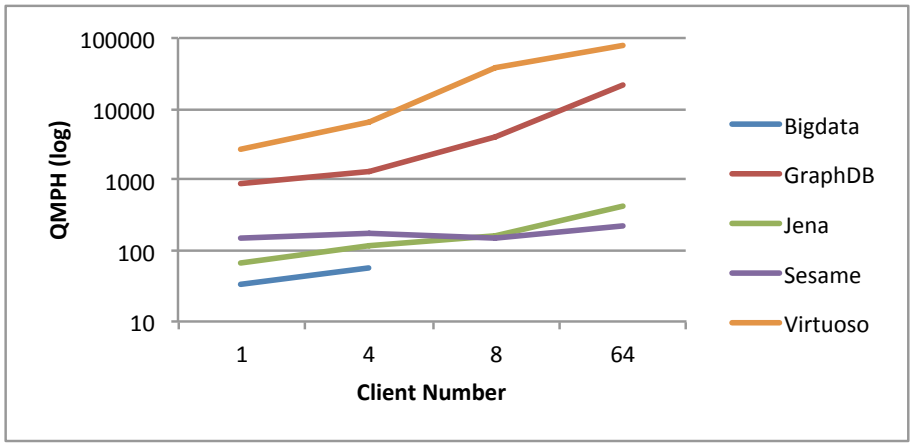


Figure 5.6: Comparing the performance of different RDF stores across different clients using the full DBpedia dataset.

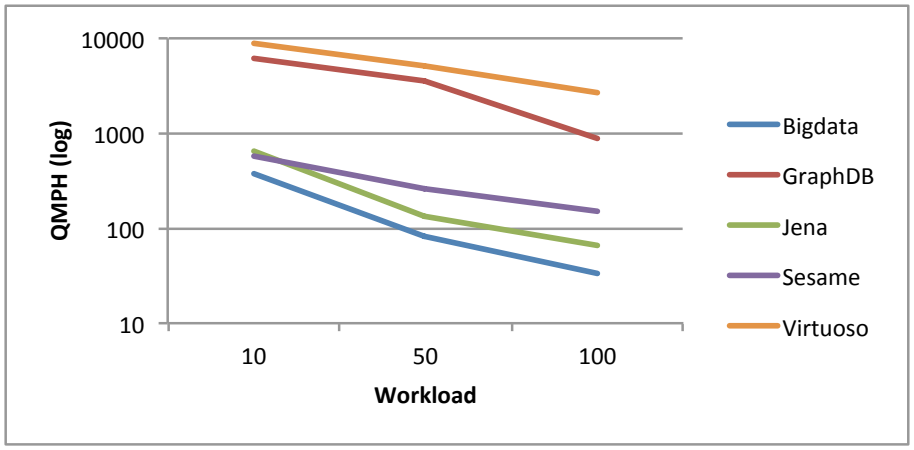


Figure 5.7: Comparing the general performance of different RDF stores with single client against different workloads of DBpedia.

5.3, clusters (6,11 and 15) causes Jena to be slower than Virtuoso by more than an order of magnitude. We also ran the same benchmark with different clients (1, 4, 8, 64) sending in parallel queries to the targeted RDF store. Figure 5.6 illustrates the QMPH for each RDF store on different clients. Bigdata only managed to perform with 4 clients at a time, and failed to return any results afterwards. Notably, GraphDB managed to keep up with the increasing number of clients, especially at the end, with 64 clients producing more than 10,000 queries per hour. Sesame, on the other hand, did not show any parallelism capability and later at the 64 clients stage flipped position with Jena, which has been steadily increasing.

It is also important to check the RDF stores' performance with smaller datasets. Since DBPedia is a real-world dataset, producing a smaller dataset (from DBpedia) is not straightforward. For this, we use the approach described by Morsey et al. (2012) to divide DBpedia into 50% and 10% subsets. Figure 5.7 also repeats the same pattern with different stores. Virtuoso and GraphDB seems to be in a cluster, while the other

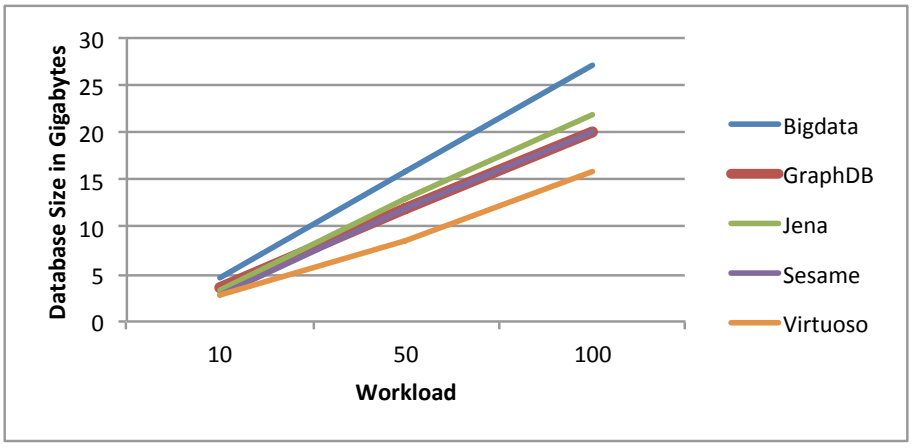


Figure 5.8: Database size with different workloads of DBpedia within each RDF store.

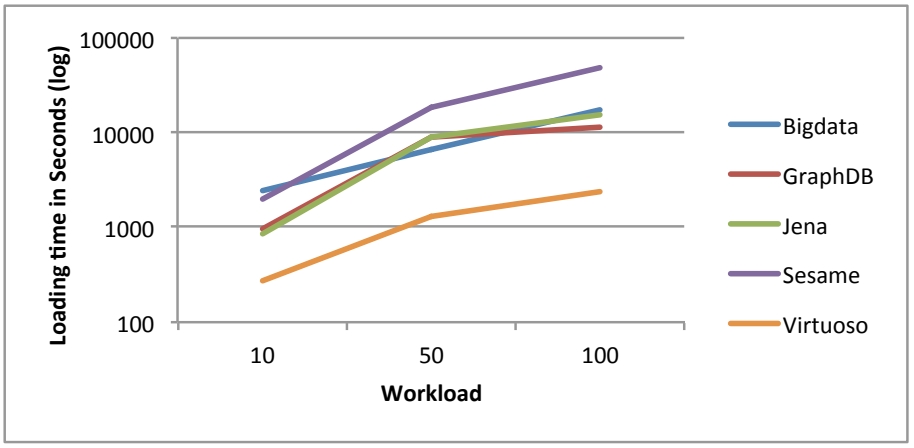


Figure 5.9: Loading time with different workloads of DBpedia within each RDF store.

three are in another cluster. Sesame and Jena also flipped position again when running the benchmark on a 10% of DBpedia. Figure 5.8 gives the resulting database size for each workload. The figure indicates that all the RDF stores grew their database in a linear fashion to the size of the workload.

Notably, the graph shows the size of the database for each RDF store has the opposite performance in QMPH in Figure 5.6. For example, although Virtuoso has the smallest database size, it was able to produce the highest queries per hour. This means that there is still room for other RDF stores to optimise their database sizes. The ratio of loading time across different workloads is also depicted in Figure 5.9. Although Virtuoso maintained a good loading time in relation to the others, GraphDB moves towards Sesame. One explanation could be because of the underlying Sesame Sail used in GraphDB.

5.7 Conclusion

In this chapter, we presented a generic SPARQL benchmark (CBSBench) that covers the other type of benchmarks: real world benchmarks. This benchmark reveals the performance differences within SPARQL features. Although there is a large number of SPARQL log queries available, previous benchmarks opted to reduce the number of the overall queries into a smaller set of queries that are meant to represent the larger set of queries. In this benchmark, all queries from the log take place when running the benchmark. We also presented our results based on clusters rather than queries. By doing so, the influence of different SPARQL features becomes more noticeable. Since the number of tested queries is large because they have slight variations from each other, the benchmark can be useful for developers to detect errors when implementing various SPARQL features in their proposed RDF stores.

We also proposed a new SPARQL clustering method that is based on a K-means algorithm with a new set of features. We compared this clustering method to a previous approach and showed that our method gives better clusters than another similar approach on three RDF stores. Finally, we reported the results of running the benchmark on five RDF stores. The benchmark tests various factors, such as workloads and clients. This benchmark can also be used to evaluate more specific SPARQL features, such as Regex-specific queries. The next chapter presents a new Regex index that can be used to optimise the performance of Regex queries within SPARQL. CBSBench will be used within the evaluation part of the next chapter.

CBSBench provides a great help to developers who work with RDF stores development. The benchmark can be used to detect implementation errors/inconsistencies within executing various SPARQL features. For example, a team of developers who are designing a new RDF store can use this benchmark against their RDF store. The result of the benchmark can be used to check the number of results for each query and compare them with a different reliable RDF store such as Virtuoso. Any differences in the number of the result set between the two RDF stores would imply implementation errors within the proposed RDF store. Moreover, the development team may also discover performance issues with particular cluster, such as: a cluster with more than one OPTIONAL clause. This could be useful to the development team to dig deeper into implementation of that clause to find out what is causing this performance issue.

Chapter 6

(Reg|Ind)ex: Regular Expressions Indexing within SPARQL

In the above chapters, the thesis has described how people use Regex within SPARQL. The analysis of people usage greatly influenced the design of the following proposal. Two different SPARQL benchmarks (BSBMstr and CBSBench) are used to evaluate the proposal. This chapter answers the final research question (4):

What optimisations can be done to increase the performance of Regex within SPARQL?

This chapter proposes a Regular Expressions Index, named (Reg|Ind)ex, which increases the performance of SPARQL queries which contain a Regex filter.

The chapter is structured as follows: Section 6.1 gives an introduction to the problems related to Regex within SPARQL. Section 6.2 discusses the related work in this area. Section 6.3 describes the design decision of the index and our algorithms for building and querying the index. In Section 6.4, we present our evaluation methodology and the results of running the experiment. In Section 6.5, some limitations about our approach are discussed. Finally, we conclude our findings in Section 6.6.

6.1 Introduction

Searching through strings has been made possible within SPARQL by using filter expressions that contain Regular Expressions (Regex) (Feigenbaum et al., 2013). However, Regex can be slow for two main reasons: firstly, it requires high computations because of their high expressivity. Secondly, it requires a full scanning of records, which results in large disk accesses (Lee et al., 2011). Figure 6.1 illustrates the native expression tree for executing a standard Regex query. This query, for example, requires all records of

the index (POS), which have their keys as "rdfs:label", to be fetched and scanned with the filter expression.

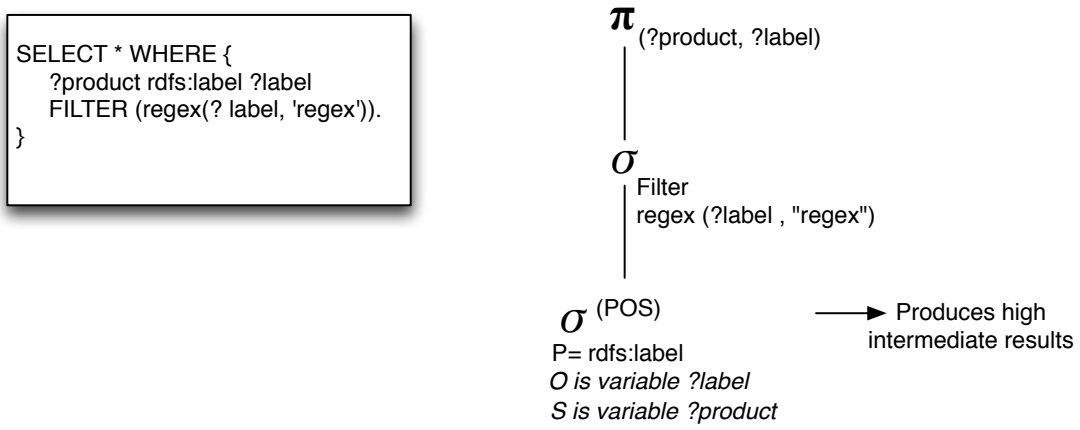


Figure 6.1: Expression tree for a native query within SPARQL.

To address the issue of having to process a large number of records, we propose a prototype named (Reg|Ind)ex, a Regular expressions index that primarily reduces the result set to be scanned with the engine. This index results in a relatively smaller index size as well as becoming portable due to our reuse of RDF store extensions for Full-Text Search. We evaluate this proposal using BSBMstr, an extension of the Berlin SPARQL Benchmark for Regular expressions and XPath queries proposed in Chapter 4. This approach outperforms the original by an order of magnitude in executing the queries, in general.

6.2 Related Work and Discussion

Indexing data has been a known approach in different topics within Information Retrieval (IR) and BDMSs for improving the performance of retrieving data, but at the cost of building certain data structures. It can also be referred to as approximate matching, in contrast to exact matching as in Regex. It is used with the relational model such as SQL, for indexing tables using `B+Tree`. Some BDMSs use Bitmap index for storing low cardinality fields, such as gender or status. In-memory BDMSs may prefer to use a hash table as their main index structure. Full-text search (FTS) engines (such as Lucene or Sphinx) usually implement an inverted index[1].

Within BDMSs, some use an integrated FTS within the original data index to provide an FTS within their querying language for the purpose of approximate matching, such as SQL with Sphinx or SPARQL with Lucene such as LuceneSail (Minack et al.,

[1] `https://lucene.apache.org/core/3_0_3/fileformats.html`

2008). PostgreSQL implements a tri-gram index to optimise the performance of similarity search[2]. Another approach to indexing the RDF data within Lucene for efficient filter expressions answer is to use a Regex engine within Lucene called FILT (Stuhr and Veres, 2013).

The notion of n-grams (including tri-grams), in general, has been discussed in a number of fields, such as biological sciences and natural languages. In natural languages, for example, a bi-gram of the phrase "regular expression indexing" would yield bi-grams, such as "regular expression, and expression indexing". Tripathy et al. (2016), for example, proposed classifying movie reviews by various machine learning techniques using n-grams, such as uni, bi and tri-grams combinations.

Regular expressions received little attention in the area of optimisations. Rather, Regex is usually discussed in a computational context, such as (Gruber and Holzer, 2014; Becchi and Crowley, 2008; Smith et al., 2008). In the context of Regex within n-grams (including tri-grams), in general, in the area of biological sciences, Paliwal et al. (2014) presented a tri-gram feature extraction technique to improve protein fold recognition accuracy. Chan et al. (2003) investigated the opposite problem in that, for a given string and a large list of Regular expressions, finds the set of Regular expressions that satisfies the given string. There are various applications that can benefit from such usage, such as XML (filtering, routing or classifications). This thesis proposes a Regular expressions tree "RE-tree" index structure that reduces the set of Regular expressions to be evaluated against the string to a smaller representative set using a tree structure that is built on top of all Regular expressions.

Cho and Rajagopalan (2002) introduced a radical approach to indexing text documents for Regex querying named FREE. Another refinement was also proposed by Chen (2012). Within the SPARQL area, we are only aware of one approach, named RegScan (Lee et al., 2011), which is also based on FREE. An improved version that supports updates has been published by Lee et al. (2015). RegScan is built using a multi-gram/n-gram approach by indexing strings using various grams. This, in a naive approach, would result in a very large storage space. RegScan uses a selectivity technique proposed by Cho and Rajagopalan (2002), which aims to reduce the number of grams to be indexed. The selectivity technique is mainly inspired by TF-IDF; the general role is that the more frequent a gram is used, the less important the gram is among others within the index. "Less useful" grams are then discarded from the index which helps reducing the size of the index. The idea of n-grams increases the querying time, because it is possible to query the index using the most selective gram, which reduces the number of intermediate results to the minimal. However, because some grams are removed from the index, this means that there are cases where the index would not be any use.

[2] `https://www.postgresql.org/docs/9.3/static/pgtrgm.html`

In this work, we experiment with a different approach: a fixed tri-gram index. It is expected that this approach would generally give a smaller index compared to a multiple one, given that the multiple/n grams within probably includes tri-grams. The size of the index is important here since some relevant RDF datasets are in the order of tens of gigabytes (i.e., the English version of DBpedia 3.5.1 is 37 gigabytes of plain N-triples format[3]). Moreover, in this approach, no selectivity measures are taken, which means that, given the existence of at least one tri-gram, it is always possible to use the index to produce some intermediate results. The risk would be then on the querying side; because, in general, some tri-grams would relatively, in general, yield a larger set of intermediate results in comparison to longer grams, such as 4-grams or 5-grams. Within such an implementation, the greater the number of intermediate results that exist, the longer it generally takes to evaluate a given query.

We also provide a framework that can be easily plugged into RDF stores, specifically, those which already provide FTS capabilities. Unlike RegScan, we integrate our approach with the overall SPARQL evaluation plan. Our evaluation is also more realistic, since we compare an RDF store before and after implementing the algorithms, unlike RegScan where they compare a prototype built using C language against Sesame, which is a production solution to manage Semantic Web technologies.

6.3 (Reg|Ind)ex Design

This section describes the design decisions regarding the (Reg|Ind)ex. It is intended for this system to be able to work within an existing RDF store. This mainly affects the logical and physical plans of the original SPARQL query plans. However, this study assumes that RDF stores are responsible for optimising these plans and, therefore, focuses on the internal processes of building and querying the index. The assumption is valid as long as those RDF stores are already using a similar utility, such as Full-text search to perform approximate matching, in which case, their plans can be extended to (Reg|Ind)ex. As a consequence, there will be a limit on the optimisations that can be done during the querying phase, and only those general optimisations can be implemented (more on this in Section 6.5).

6.3.1 Index Building

Regular expressions can be used on both literal strings and URIs. In this proposal, we only consider improving the efficiency of evaluating literal strings. Firstly, we formally define RDF triples, since it is the starting point of building the index.

[3] `http://downloads.dbpedia.org/3.5.1/en/`

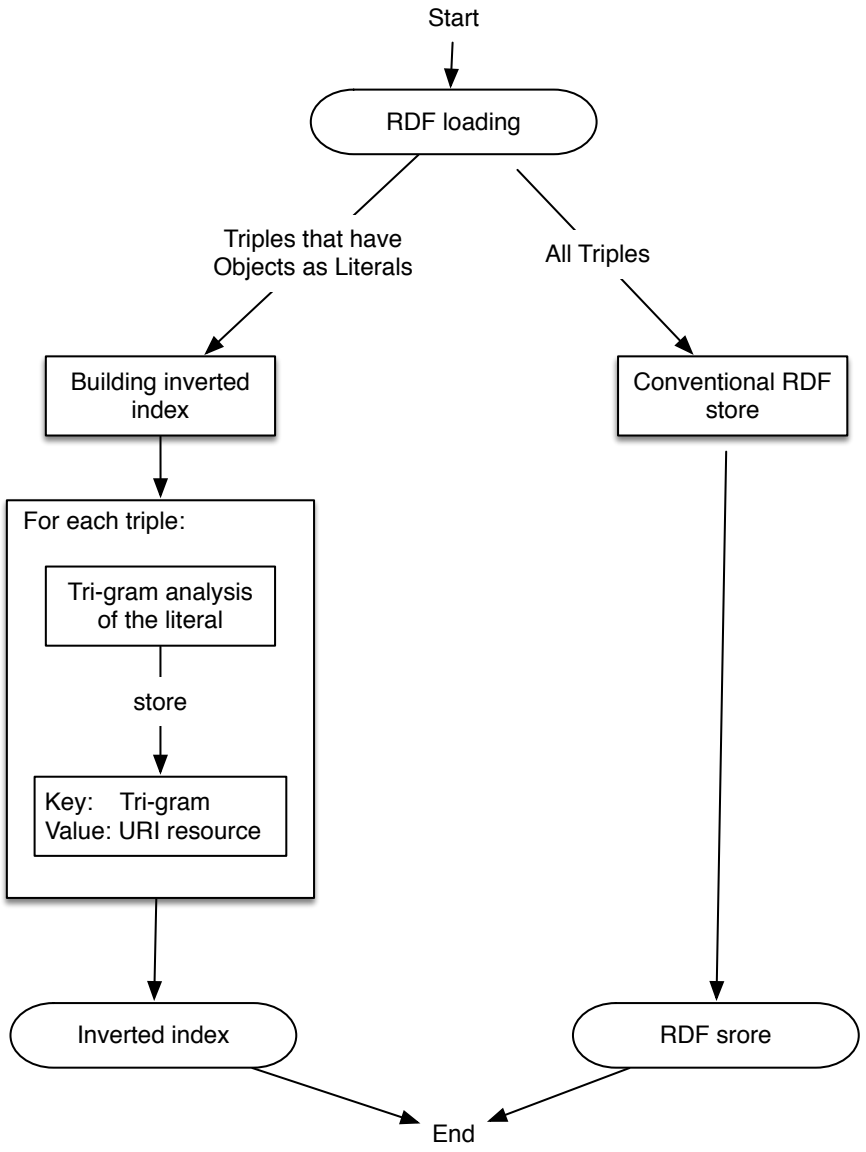


Figure 6.2: (Reg|Ind)ex Building architecture diagram.

Definition 7. (RDF triples) Given an RDF document which is a set of RDF triples tr, a triple $(v1, v2, v3) \in (U \cup B) \times U \times (U \cup B \cup L)$ is referred to as an RDF triple tr where U is a URI, B is a blank node and L is a literal value.

Definition 8. (String Length and Substitute) Given a string s, the function $length(s)$ returns the length of characters in s. Function $substitute(s, x, y)$ returns a portion of s where x is the starting position of the character in s, and y is the ending position of the character in s.

Figure 6.2 alongside Algorithm 2 illustrate that our building phase can be triggered to take place in parallel, while RDF triples are inserted into an RDF store. For each triple, if $v3 \in L$, then the index is built.

Algorithm 2: (Reg|Ind)ex Index Building

Data: RDF document.
Result: Inverted Index: I[T].
Initialisation;
there ia a unique id for each $v3 \in L$.
foreach *tr (v1,v2,v3)* **do**
  if $v3 \in L$ **then**
    $T \leftarrow$ Extract Tri-grams($v3$) # (presented in Algorithm 3);
    foreach $t \in T$ **do**
      if $I[t] \neq nil$ **then**
        $I[t] \leftarrow I[t] \cup \{id(d3)\}$;
      else
        $I[t] \leftarrow \{id(d3)\}$;
      end
    end
  end
end

Algorithm 3: Extract Tri-grams

length(s): it returns the length of characters in s.
substring($s, start, end$): it returns a portion of the string s where $start$ is the starting position of the character in s, and end is the ending position of the character in s.
Data: String s.
Result: Set of Tri-grams T.
Initialisation;
for $i = 0$ ***to*** $i \leqslant length(s) - 3$ **do**
  T $\leftarrow T \cup substring(s, i, 3 + i)$
end
return T

In RegScan (Lee et al., 2011), building the index is relatively more resource intensive. This is because of the selectivity technique that is being used. For example, redundant grams have to be removed, which requires statistics to be gathered about the grams and their frequencies. In this approach, data is directly inserted into the index iteratively. This makes our approach capable of loading a large set of data.

It is important to construct the index structure efficiently to handle a large class of pattern searching, such as Regular expressions. This approach relies on the notion of grams, specifically tri-grams.

Definition 9. (Tri-grams) Given a string s with a length $len(s) \geq 3$, tri-grams tri are all the sequent substrings of characters of s where the length of each tri-gram $len(tri) = 3$.

For example, a string such as "Regular Expressions" should yield tri-grams such as {*'Reg', 'egu', ..., ' Ex', ..., 'ons'*}, noting that even the space will be counted as a character. Moreover, it is possible to denote both Start-with and End-with in a literal

by a special symbol; initially, we chose underscore "_" at this stage. Therefore, from the example above, we can add {*'__R', _Re, ns_, s__*} to represent both cases.

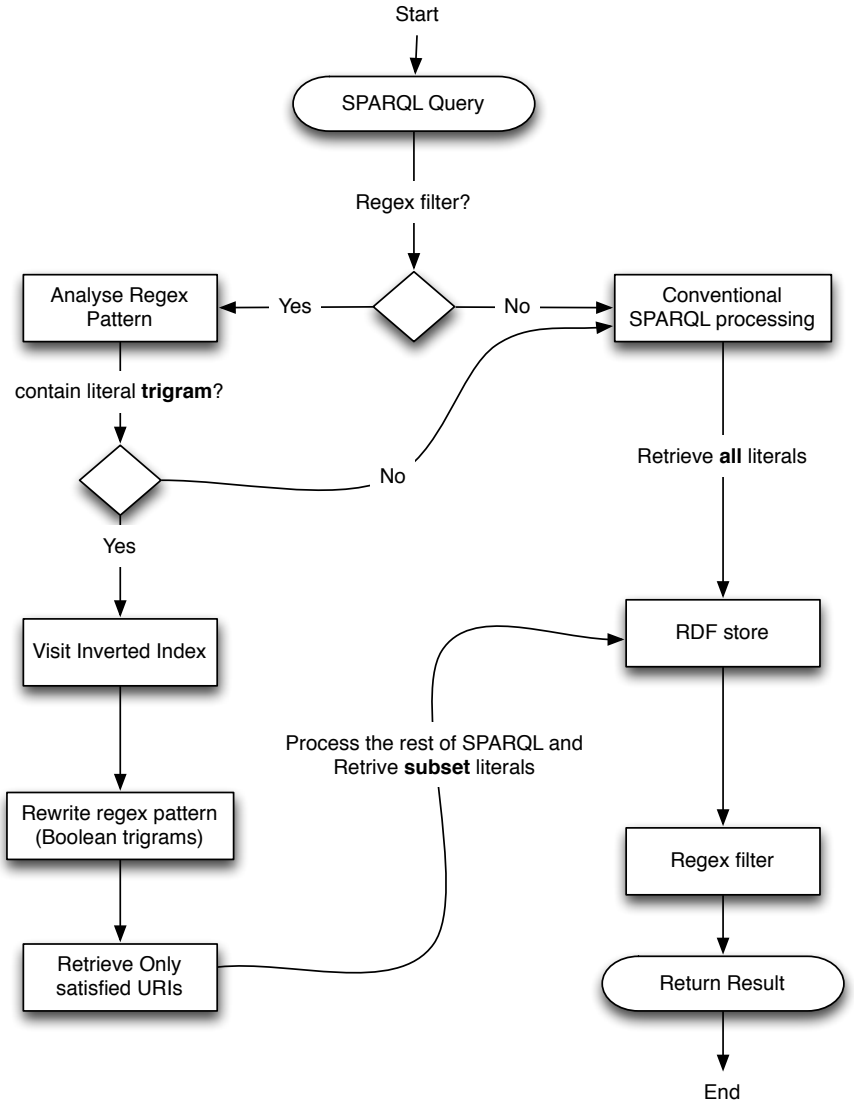


Figure 6.3: (Reg|Ind)ex querying architecture diagram.

6.3.2 Query Processing

Figure 6.3 shows the overall architecture of SPARQL queries' execution. This approach is triggered by a SPARQL query that includes a Regex filter expression. Then, there will be a high level analysis of the Regex pattern before applying any further optimisations. This mainly performs a linear scan of the Regex pattern searching for any tri-gram sequences. The process of executing a SPARQL query is described by Algorithm 4.

Algorithm 4: (Reg|Ind)ex SPARQL Querying

default() function refers to the default procedure of the RDF store;
Data: SPARQL query Q.
Result: Result Set RS
Initialisation;
if *Q includes Regex Filter RF* **then**
  default(execution);
else
  if *RF does not have any tri-gram T* **then**
    default(execution);
  else
    default(build execution plan);
    ParseTree $PT \leftarrow$ Parse Tree Construction of RF;
    Result Set $RS \leftarrow$ Query Inverted Index I[T] using BT;
    default(continue executing plan);
  end
end
return RS

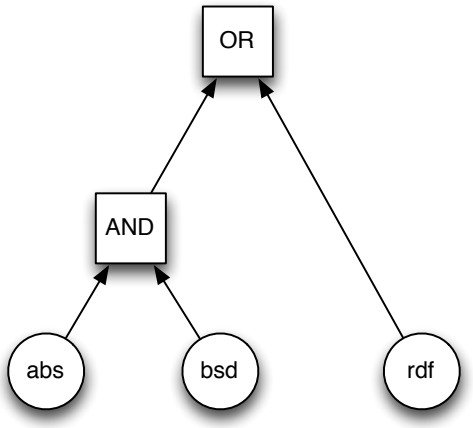
6.3.2.1 Parse Tree

This technique is inspired by the work of Cho and Rajagopalan (2002). To be able to construct a parse tree, we first revise the operations of Regular expressions. There are three main operators within Regex.

- Concatenation: if A and B are Regular expressions, then AB is also a regular expression.
- Alternation: if A and B are Regular expressions, then $A \mid B$ is also a regular expression.
- Kleene Closure: if A is a regular expression, then A^* is also a regular expression.

It is then possible to construct a parse tree by only using "AND" and "OR" operators. The AND operator reduces the number of tri-grams to be returned; there will be less false-positive results. The OR operator is rather necessary to ensure the usage of alternations within the regular expression; it ensures there will be no false-negative results. For example, the Regex pattern "absd|rdf" is converted into a parse tree and query. Figure 6.4 depicts the parse tree output of both tree and query. Another more complex Regex pattern is "(rdfg|esd).*wsdk" which is also depicted in Figure 6.5.

Regex patterns, such as "absd|rdf" or "(rdfg|esd).*wsdk" are translated into an internal tree, as shown in Figures 6.5 and 6.4. All leaves represent a sequence of three normal characters (tri-grams). Other meta/special characters which have special meanings within Regex are eliminated such as ".*". This also means that it is possible to




```
Parse Query: ( ( abs AND bsd ) OR rdf )
```

Figure 6.4: Regex Pattern "absd|rdf" to a parse tree.

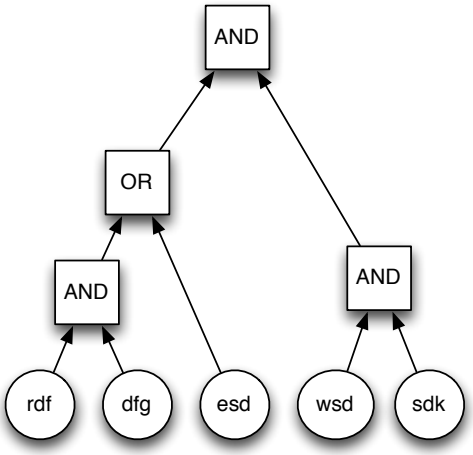
have the same parse tree as that from Figure 6.5, when applied to the Regex pattern "(rdfg|esd).*[0-9]wsdk", or even to the Regex pattern "(rdfg|esd).*wsdk.*a" even though the last character "a" is not a tri-gram and, would be subsequently eliminated.

The parse tree is then executed against the inverted index, producing a set of posting lists compliant with the parse tree. The syntax of the parse query is compatible and executable by Lucene Framework[4].

Negation, however, is not included in this implementation. For example, the Regex pattern "∧(William)", which matches a string, which does not contain the word "William" can not be evaluated against the current implementation of (Reg|Ind)ex; rather, it would have to be evaluated with the default Regex evaluator by the RDF store and would cause a full scan of all intermediate results. The other main type of Regex pattern that does not satisfy the conditions of (Reg|Ind)ex is having no tri-grams within the pattern. For example, a Regex pattern, such as "ad.*rs.*[0-9]{5}.*d?" cannot be parsed into a binary tree because it does not have any consecutive three literals within the Regex pattern.

Figure 6.6 gives an example of how it is possible to query the inverted index in which keys are a set of tri-grams and values are a set of IDs that represent pointers to actual literals.

[4] `https://lucene.apache.org/core/3_0_3/api/core/org/apache/lucene/search/BooleanQuery.html`




```
Parse Query:
( ( ( rdf AND dfg ) OR esd ) AND ( wsd AND sdk ) )
```

Figure 6.5: Regex Pattern "(rdfg|esd).*wsdk" to a parse tree.

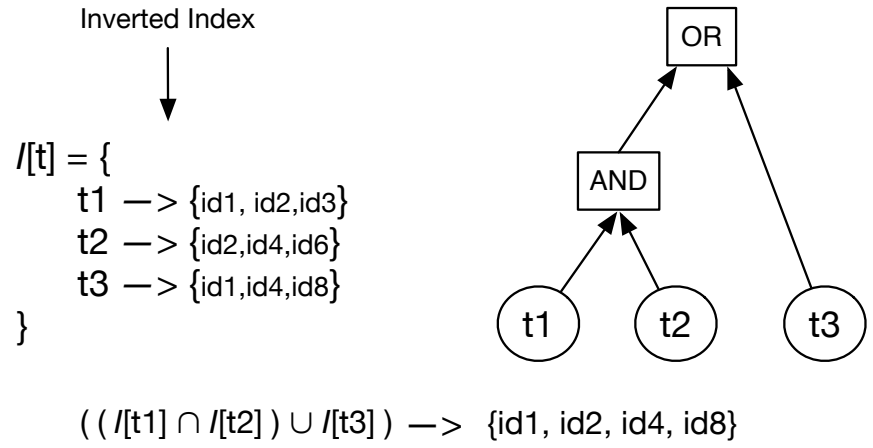


Figure 6.6: An example of querying the inverted index over a set of tri-grams.

6.3.2.2 Join Operations

There are two main joins included within this approach. Firstly, when the parse tree is executed against the inverted index, there will be joins between the posting lists to produce all pointers satisfying the parse tree. This is the first join operation within the inverted index itself. Secondly, a new join operator should be built on the level of the SPARQL query, which ensures what parts of the BGP are relevant to the intermediate results produced by the inverted index.

SELECT * WHERE {

```
    ?product rdfs:label ?label
    FILTER (regex(? label, 'regex')).
}
```

Listing 6.1: SPARQL query within Regex filter example.

Figure 6.7 shows how the new expression tree is constructed for the query in Listing 6.1. Instead of having to fetch a large number of records, in this case, from the index (POS), a join can be applied prior to the filter scan which joins the relevant tri-grams according to the Regex pattern included within the SPARQL query with the index (POS), which should reduce the intermediate result set.

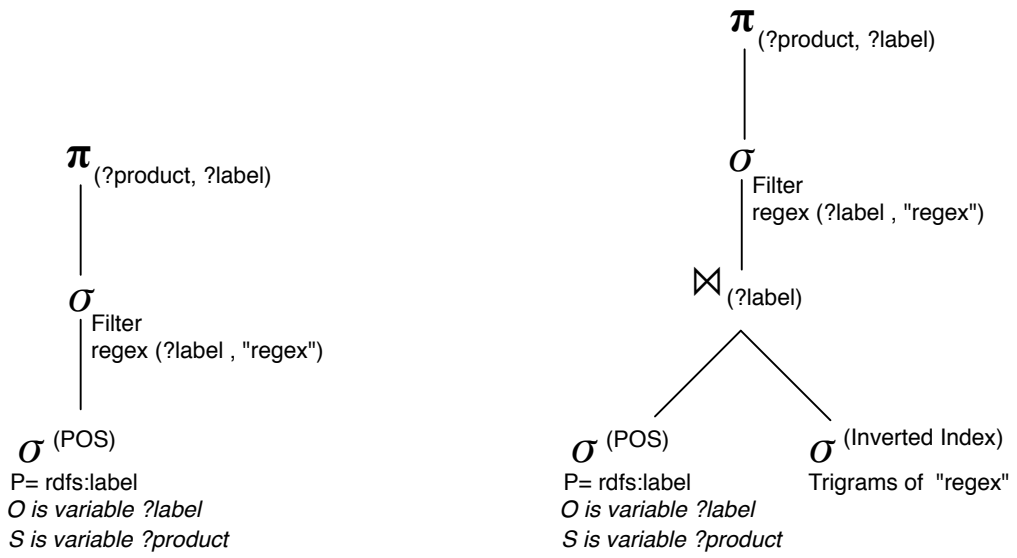


Figure 6.7: Comparison of both the native and the proposed expression tree.

6.4 Evaluation Methodology

This section presents the process of evaluating the proposal to apply the tri-gram index to increase the performance of processing SPARQL queries when a Regex pattern is being used within the query. Section 6.4.1 describes the rationale behind the design decisions to implement a tri-gram index. In Section 6.4.2, we present the benchmarks we use to assess the performance of SPARQL queries, in which the queries are used to test RDF stores with, and without, a tri-gram index being used. The raw queries and data are presents in detail within Appendix D.

6.4.1 Tri-gram Index Implementations

To be able to test this proposal, a modification to the chosen RDF stores has to be implemented. This is to make the RDF store deal with our Regex index. We do not modify any internal logic, we only add the proposed index to the code base. Then, we add a SPARQL keyword which would trigger the query optimiser to use our index instead of the original work-flow. The keyword we use in the case of Jena is: "text:regex", whereas in Sesame: "rev:text"as well as some other keywords. Sesame has slightly different original implementation and we had to adapt to their original implementation. Check Appendix D for some examples of SPARQL queries that has been used for querying our proposal.

Moreover, it is preferable to make use of other ready solutions, which not only save time, in contrast to building a tri-gram index from scratch, but also reduces implementation errors, when a robust and well-tested solution is being used. It seems that most of the well-known RDF stores such as Jena, Sesame, Bigdata and some others (Minack et al., 2009), already implement a full-text search index, as an extension to perform approximate matching, to enable users to search through text within SPARQL. This extension is still not part of the SPARQL standardisation (Aranda et al., 2013). Both Jena and Sesame are chosen for this proposal's adaptation, as they both have suitable licenses and implement a full-text search facilities, as well as having an active support community to help developers with their questions on the web [5][6]. The general idea is to utilise the full-text index/query to be used as a tri-gram index/query model.

Some RDF stores have also adapted other ready implementations for their full-text search capabilities. Both Jena and Sesame have used Apache Lucene[7] as their back-end full-text search implementation. We modified their interfaces to implement a tri-gram index/querying. We also rely on their method of querying the index, mainly by using an extension predicate within the SPARQL query language. This means that we did not modify any logical or physical execution plans, leaving them at the RDF store default settings. This can cause performance variations, depending on how well the default full-text query plan implemented by the RDF store. To overcome this issue, we compare each RDF store performance with and without the index, but avoid comparing RDF stores with each other. Figure 6.8 shows the result sizes of building the database within the BSBM dataset, which has 8 million triples "bibm8" as named by BSBM team, and the full dataset of the English version of DBpedia within two different RDF stores: Jena and Sesame. This figure indicates that Jena builds a relatively smaller (Reg|Ind)ex than Sesame. The overall relation of the resulting (Reg|Ind)ex to the original database size within bibm8 is 35% and 14% within Sesame and Jena, respectively, and 16% and 10% in DBpedia.

[5] https://jena.apache.org/help_and_support
[6] https://groups.google.com/d/forum/rdf4j-users
[7] https://lucene.apache.org

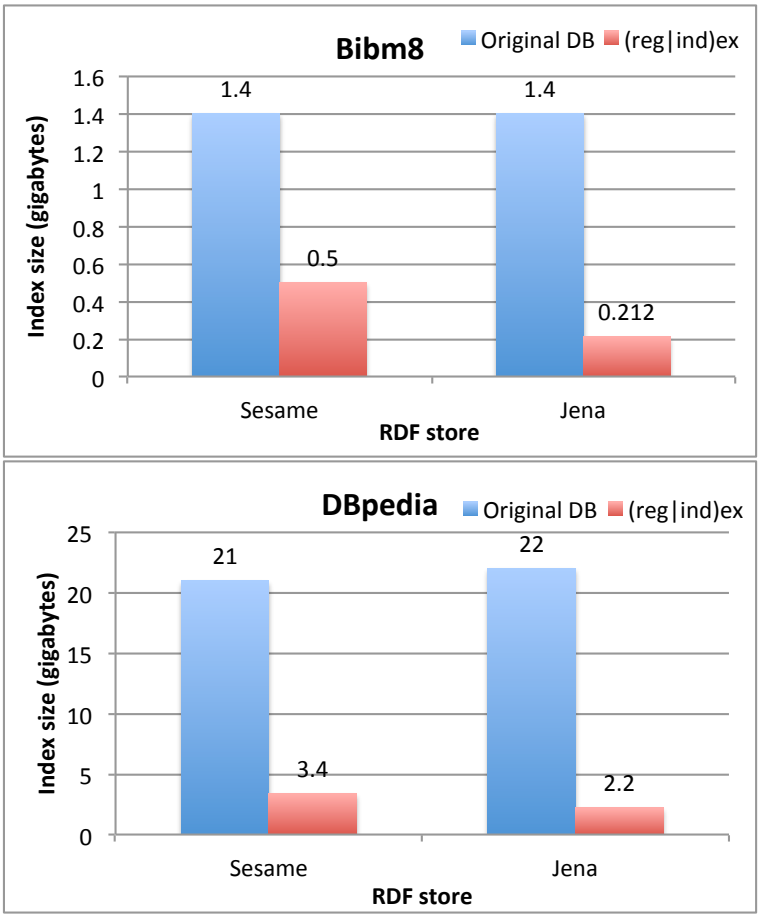


Figure 6.8: The size of both the original database and index sizes within Jena and Sesame RDF stores on Bibm8 and DBpedia.

RegScan apply the GeneOntology RDF dataset [8] in their evaluation. The dataset is considered to be a real dataset (similar to DBpedia, in contrast to a synthetic one, such as the BSBM dataset). That version of GeneOntology contains 869,770 triples compared to 8 million of BSBM triples and around 220 million triples of the English version of DBpedia. In RegScan, the index size of the GeneOntology dataset is around 500% to the size of the original Sesame database size.

This is a huge difference between the two approaches, which might be caused by two main reasons. Firstly, RegScan implements an n-gram approach with a minimum and maximum threshold (2,4), which yields three grams as: 2, 3 and 4. As far as building indices is concerned, multiple grams would greatly affect the size of the index, in comparison to a single gram (tri-gram), as in our approach. The second reason is that RegScan does not only build the index literal values (as within our case for bibm8), but also builds grams on all triples including subjects, predicates and objects. This allows RegScan to be able to facilitate a wider spectrum of Regex queries, unlike (Reg|Ind)ex with bibm8, which only can answer those queries that conform to the design described

[8] http://www.geneontology.org

in Section 6.3.1. Such a design requires the RDF triple to have objects as literals. This design decision was inspired by the analysis of how people are actually using Regex within SPARQL queries (see Chapter 3).

6.4.2 Evaluation Benchmarks And Results

This section describes the evaluation methods and results relating to the (Reg|Ind)ex performance. The general idea is to compare an RDF store performance before and after (Reg|Ind)ex is being adopted. To this end, the evaluation is done by using two different Regex-specific benchmarks, (BSBMstr and CBSBench) presented in chapters: 3 and 5. BSBMstr has two test suites: Regex and XPath. In this evaluation, only Regex queries will be used, as XPath ones are irrelevant to this evaluation. In the case of CBSBench, only small additions have to be applied, as CBSBench is a generic SPARQL benchmark. While we intend to test only those queries with Regex filters, more Regex-specific features would give a better explanation about their performance.

For both BSBMstr and CBSBench, we initially run the original implementations of RDF stores before, and after, embedding (Reg|Ind)ex, to ensure that both implementations have consistent result sets; they both should produce an equal number of results. Those queries which are not consistent are removed from queries. In the case of BSBMstr, all queries are consistent between the two approaches. CBSBench has been mainly affected by this step, due to the high variety of queries within the benchmark. This step is important because (Reg|Ind)ex is not expected to capture all Regex patterns' cases, especially those with highly nested brackets or escaped characters.

The machine we used to run our test is a dedicated machine that has the following specifications: 2x AMD Opteron 4280 Processor (2.8GHz, 8C, 8M L2/8M L3 Cache, 95W), DDR3-1600 MHz 128GB Memory for 2CPU (8x16GB Quad Rank LV RDIMMs) 1066MHz 2x 300GB, SAS 6Gbps.

6.4.2.1 Evaluating (Reg|Ind)ex with BSBMstr

We implement the methods described in Chapter 4. The part of that chapter related to XPath is not implemented. Moreover, we modified SPARQL queries to have the trigram predicate extension when testing (Reg|Ind)ex, while keeping the original queries from the original RDF store. We perform the test on the BSBM dataset that has 8 million triples. The size of the database without the index for both Jena and Sesame is 1.4 gigabytes. The index size for Jena is 212 megabytes and 500 megabytes for Sesame.

Figures 6.9 and 6.10 show a general comparison between (Reg|Ind)ex and the original implementations of two different RDF stores: Jena and Sesame. In general, (Reg|Ind)ex almost always outperforms the original implementations of both RDF stores by an order

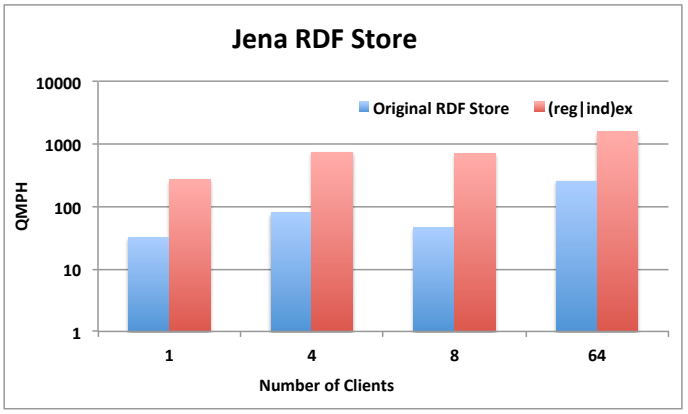


Figure 6.9: Comparing (Reg|Ind)ex and the original implementation of Jena with different clients by Regex-specific BSBMstr.

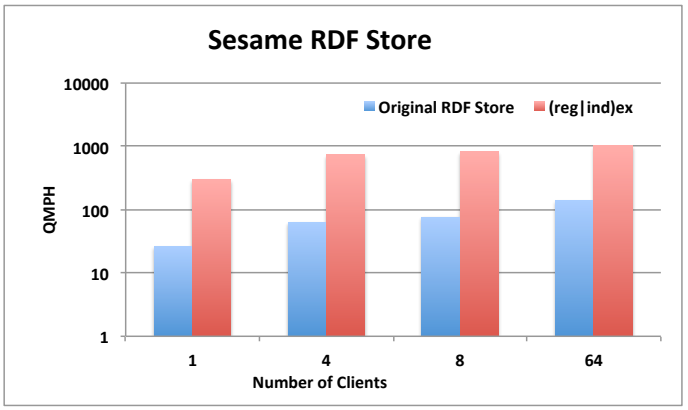


Figure 6.10: Comparing (Reg|Ind)ex and the original implementation of Sesame with different clients by Regex-specific BSBMstr.

of magnitude. Figure 6.9 indicates that (Reg|Ind)ex helps to boost Jena's performance from dropping down and maintained a steady performance, particularly when the number of clients is "8".

Figure 6.11 shows a more detailed comparison of each query using a QPS metric. Queries 3, 4, 2 and 1 respectively show that the larger the number of records and the longer the length of literal strings, the better (Reg|Ind)ex becomes against the original RDF store. However, query 3 shows that the original Regex is even better than our proposal, as "foaf:name" has a low number of records compared to others and has a shorter length. This is an interesting finding and a solution has been suggested in the future work, Section 7.1.4. Queries 4 and 5 similarly test Regex complexity, but with different predicates. Although these predicates have the same number of records, they have different average lengths. These queries show how the original method is affected by about an order of magnitude. Within (Reg|Ind)ex, the effect is minor between the two queries, but with better performance for both queries in relation to the original RDF store. Queries 5 and 6 show a test of the same predicate but with different Regex pattern complexities. Within the original RDF store, the two queries show a low and identical

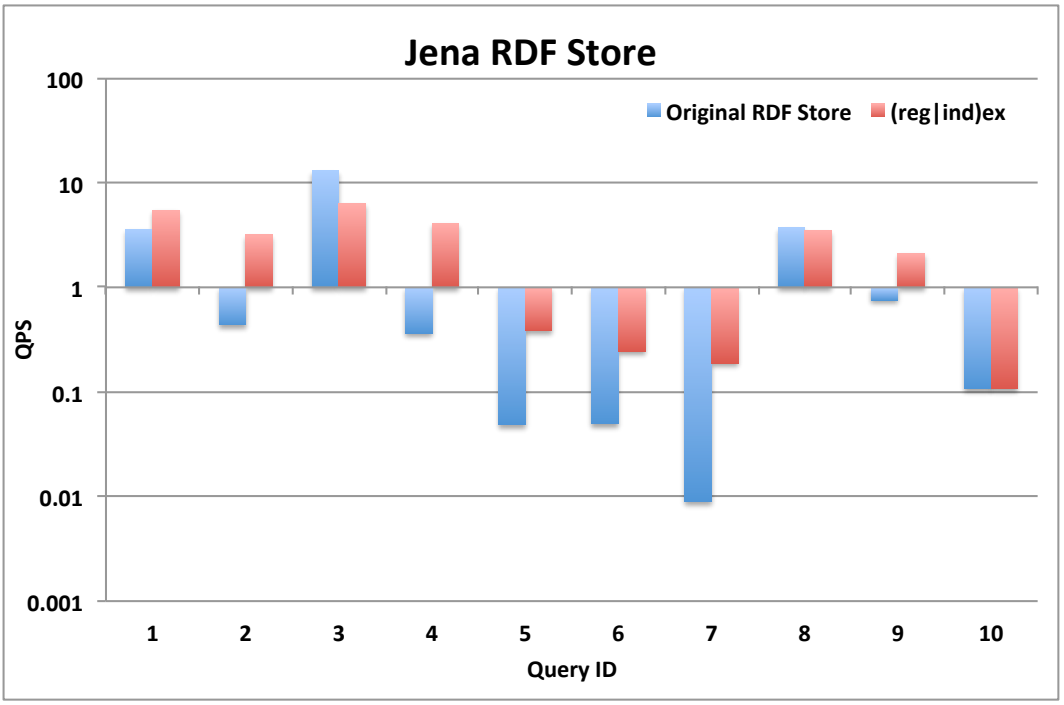


Figure 6.11: Comparing Regex-specific BSBMstr queries in both (Reg|Ind)ex and original implementation using Jena RDF.

performance. In (Reg|Ind)ex, the performance of Query 6 is slower than for Query 5 by almost an order of magnitude. This is because of the long Regex pattern that being used, which include three different generated keywords with alternations between them. This caused very long tri-gram instances to be evaluated and matched against the index. Query 7 has gained a noticeable improvement by more than an order of magnitude. However, because our index is meant to reduce the result set being scanned, the quantifier backtracking is still expensive; making this query the worst performance in the two approaches. Query 8 shows an identical performance of the two approaches; start-with Regex patterns are one of the least complex pattern, in that the length of the literal string does not affect the complexity of evaluating Regular expressions. Therefore, the time that our approach spends checking the index is identical to the time that the original RDF store spends scanning the records. Query 9, which tests the end-with property, on the other hand, is not internally executed within Regex as start-with, that is to say, it requires traversing the characters until the end. Therefore, by reducing the result set using our index, the performance of this query is improved by three factors.

Query 10 is not tested by our index. This approach requires that, for a given Regex pattern, there have to be at least three continuous normal characters, which is not the case in the pattern: “`(\b[a-zA-Z]+\b).*\\1.*\\1`”.

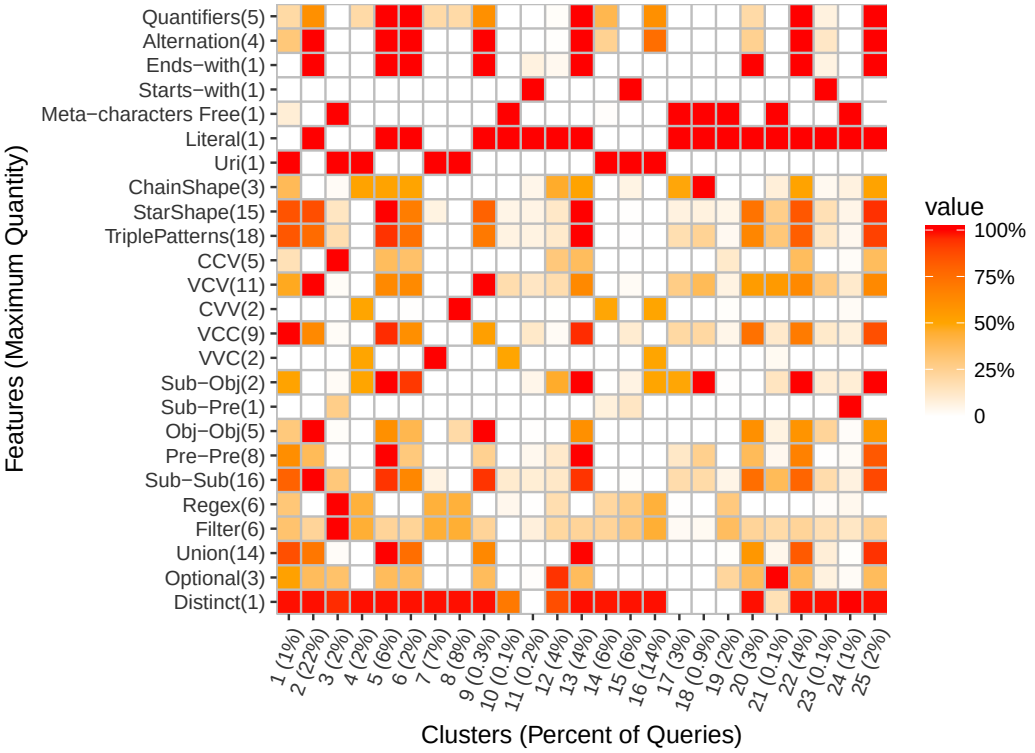


Figure 6.12: A heat map showing the existence of different Regex feature ratios to their maximum within different clusters (Note that the minimum quantity of each feature is zero except Filter, Regex, TriplePatterns and LongOfChain are equal to one).

6.4.2.2 Evaluating (Reg|Ind)ex with CBSBench

Chapter 5 presents a benchmark which measures the performance of different RDF stores against set of clusters. Each cluster represents a collection of SPARQL queries that share specific features. Those features represent a wide range of SPARQL capabilities, such as Query Type, Joins Representations and many others (see Section, 5.4.2). It is also possible to use CBSBench to evaluate the performance of (Reg|Ind)ex. CBSBench may reveal more details about the differences between the various features that would affect the efficiency of (Reg|Ind)ex. CBSBench has already shown that BSBMstr does not simulate real-world scenarios in terms of the variety of queries and how they affect various caching strategies which maybe used by RDF stores, for example. Moreover, in the original BSBM, the size of the index has not been analysed within different sizes to check how the size would grow with more triples. Original BSBM uses a dictionary/word list which has 89,523 keywords to be randomly inserted with different combinations as literals within the generated RDF dataset. This affects the size of the inverted index, in which, it that would be limited to the word list content. This causes the inverted index to only grow its posting lists, but with fixed dictionary elements.

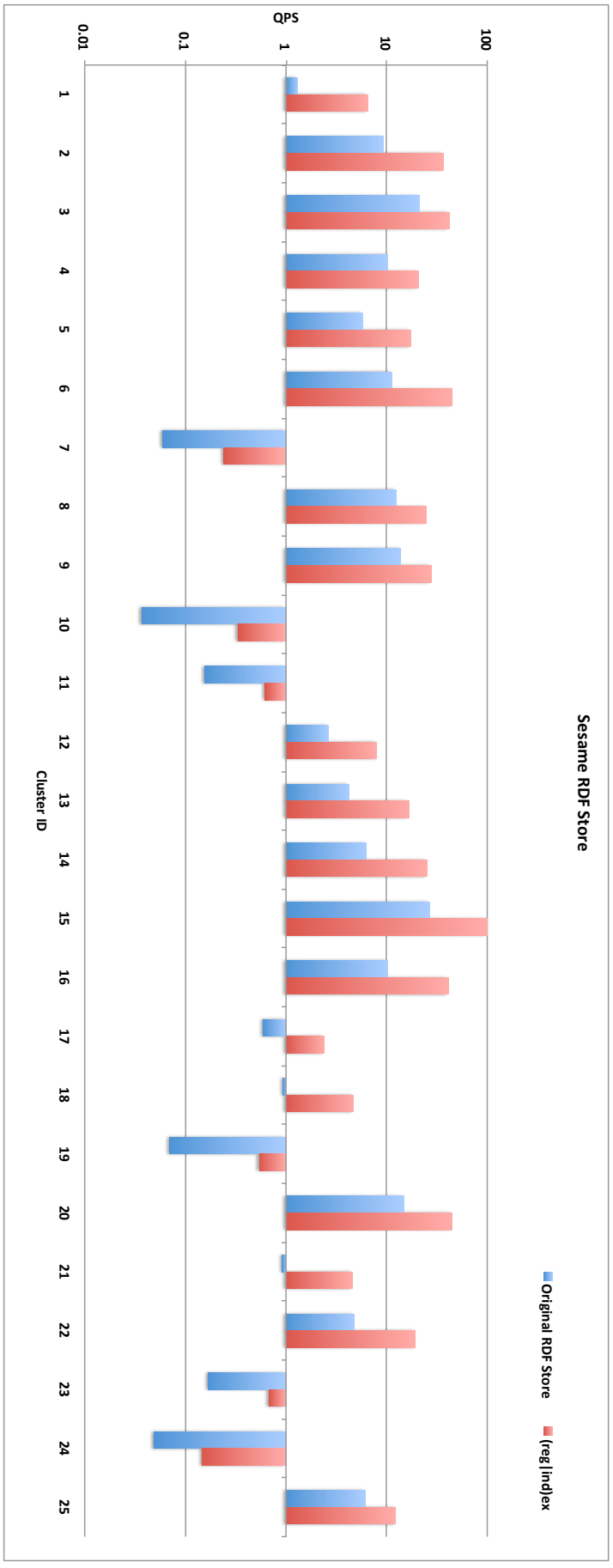


Figure 6.13: Comparing Regex-specific CBSBench queries in both (Reg|Ind)ex and original implementation using Sesame RDF.

Here, we modify CBSBench to also include some more Regex-specific features. We, however, restrict queries to only include the SELECT query type. This is not only to be consistent with BSBMstr, but also to be able to reason about the relationship between certain clusters and their performance. The role of thump within CBSBench is that the fewer the number of features being contained within a cluster, the easier it is to understand why a particular cluster is slower than others, for example. Below, there is a brief description about each of the new features being included with CBSBench:

- Uri: The Regex pattern searches through a URI. This can be checked by the predicate as they are either a URI or literal.
- Literal: Opposite to Uri.
- Meta-Characters Free: True if the Regex pattern does not contain any Regex meta-characters, such as"$* + |$" or others (we refer to the Regular Expression Syntax [9])
- Starts-with: A special case of Regex patterns that only have "^" at the start of the Regex pattern.
- Ends-with: A special case of Regex patterns that only have "$" at the end of the Regex pattern.
- Alternation: A Regex pattern that contains a number of alternation/or, such as "keyword1|keyword2" which have a maximum of one alternation. The alternation can be as many "|" as are presented within the pattern. In this case there is a maximum of four alternations (See Figure, 6.12).
- Quantifiers: The same Alternation criteria applies, while having the quantifiers operators instead of Alternation. Those include "$+, *$ and ?".

After applying query filtering and selection (as described in Section 5.4.1), as well as restricting the query result set to be equal between the two approaches, the overall queries to be clustered and then evaluated using this benchmark is 23,310 queries which is around 2.3% of the 1 million Regex queries. The vast majority of queries we eliminated because they are identical. Then we also eliminated those which have Regex patterns with no tri-gram. To check the distribution of these queries against each cluster, see the x-axis in Figure 6.12. The figure shows each cluster and its percentage of queries against the overall number of queries. For example, cluster (5) has 6% of queries, which means it has around 1,400 queries.

Figure 6.14 describes the general performance differences between (Reg|Ind)ex and the original approach, by looking at the first pair, which refers to a comparison between the

[9] `http://www.w3.org/TR/xpath-functions/#regex-syntax`

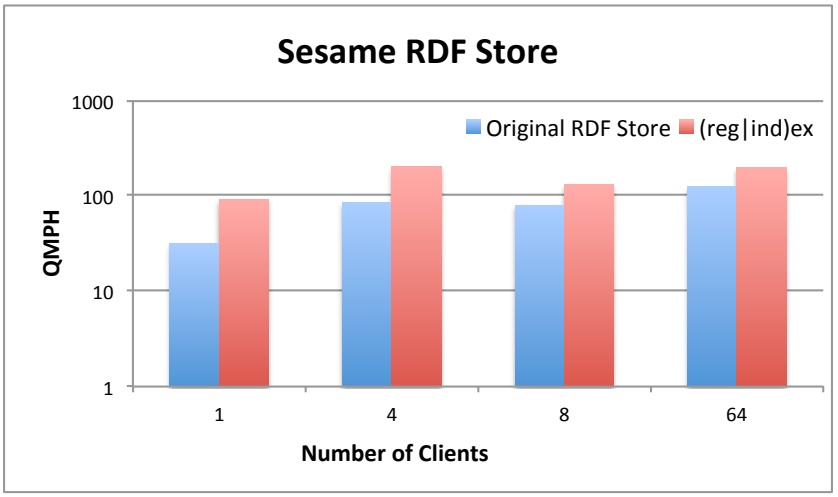


Figure 6.14: Comparing (Reg|Ind)ex and the original implementation of Jena with different clients by Regex-specific CBSBench.

two approaches using a single client/source. (Reg|Ind)ex outperforms the original implementation with Sesame RDF store, in that (Reg|Ind)ex was able to perform around 90 QMPH in contrast to 31 QMPH in the case of the competitor. The increased performance has not to expected efficiency. One reason could be that (Reg|Ind)ex includes many more queries, which puts more stress on both RDF stores and (Reg|Ind)ex. The same figure also describes the differences when using multiple clients. It appears that (Reg|Ind)ex with Sesame manages a close to linear performance against the chosen client numbers. However, the two approaches decrease at the 8 clients bar. This behaviour has already been noticed with Sesame in general (See Figure 5.6).

Figure 6.13 shows a comparison between the two approaches against a set of clusters. This figure, alongside Figure 6.12 have to be taken into consideration when reasoning about each cluster's performance. For example, Cluster (10) has increased by an order of magnitude, which seems to have been caused by the Regex patterns Meta-Characters Free and by having Literals, as literals tend to be, in general, shorter than URIs. The same case applies to Query 19. Queries 11, 15 and 23 shares a similar Regex feature (Starts-with) and they resemble the performance differences when (Reg|Ind)ex is applied.

6.5 Limitations

The problem that we tried to solve is an optimisation problem, which mostly requires trade-off decisions. In this approach, we aimed to reduce the result set to be scanned by the Regex engine used by the RDF store. This means that the index will not contain the original records but, rather, pointers to data on the original indices provided by the store. This is because in this approach there are at least two indices to fetch the original data as well as the Regex index. A worst case scenario would be if both of these

indices were not being cached. However, we used a system, Lucene, where the index is cache-friendly. If the original data was not cached, there would be an effect on both our approach and the original RDF store as well. The rule of thumb is that the larger the number of records to be scanned, plus the more complex the Regex is, the better our approach is expected to perform in comparison to the original approach. This means that for a smaller amount of data, in some cases, our index may not perform better than the original RDF store, as (Reg|Ind)ex shows. Moreover, Regex patterns with less than three contentious characters will not be able to benefit from the index, and they would require a full scanning of records.

At this stage, the SPARQL query that can benefit from the index would have to be written with a special predicate which we instantiate as an extension. Therefore, these queries are not fully SPARQL standardised. A better approach would be to keep the standard of SPARQL with a Regex filter expression; then, a query rewrite can take effect. Implementing such approach was not part of this study and is left as a future plan.

6.6 Conclusion

Querying Regex within SPARQL has been known to be inefficient. This is because of two main reasons: all tested records have to be fetched from the hard disk, which is an expensive operation, due to a rotational disk latency. Moreover, evaluating Regex is also known to be computationally expensive, because of the high expressivity attached within Regular expressions. Our approach, (Reg|Ind)ex, aims to build an index to reduce the result set to be scanned, which could mitigate both of the two reasons for inefficient Regex evaluation. Our main design decision was to use a tri-gram tokeniser. This resulted in a relatively smaller index in comparison to other approaches which have adopted a multi-gram index. Our evaluation methodology was based on comparison of a system before, and after, adopting this approach. We ran our experiment using BSBM, a known benchmark test driver, which we extended in Chapter 4. The experiment showed that our approach out-performs the original RDF store implementation of evaluating Regex in most cases.

Chapter 7

Conclusion and Future Work

The Semantic Web promises that the web would be dramatically improved in the way we query and manipulate data. The use of the RDF data model and its SPARQL querying language are critical requirements in fulfilling this vision. However, there are various performance issues with these technologies that have been discussed across this study. The performance of processing Semantic Web data represented in RDF using the standardised query language SPARQL is not satisfactory. The notion of triples, adopted by both RDF and SPARQL, increases the unpredictability of optimising queries, in comparison to SQL, for example. Moreover, RDF stores are still considered to be new in the landscape of databases.

On the other hand, Regular Expressions have a long standing history in the area of computer science. They are, however, also known to be computationally expensive, due to their linearity in matching a given string. Regular Expressions are used in many domains, such as inspecting viruses or spams on network traffic, searching phrases on the web, or finding similarities within DNA sequences. They are also used to search strings within BDMSs. SPARQL standard has added Regex with its first recommendation working group. They are meant to be used to filter results based on a given Regex pattern. Previous work and initial testing indicate that the performance of Regex within SPARQL is also not satisfactory compared to most other SPARQL features. Moreover, given the available benchmarks in the area of RDF and SPARQL, very little has been done to test Regex. The idea of this study was to thoroughly investigate the performance of Regex within SPARQL on RDF stores.

In this thesis, we first approached the problem by looking at how people, in real-world situations, use Regex within SPARQL. Thanks to the publishers of some RDF stores, people's SPARQL queries have been made available to the public to be used and analysed. These query logs show that Regex is being used by around 3% of queries within DBpedia, for example. The analysis also indicates that the Regex patterns tend to be relatively long (i.e., DBpedia average string length is 51 characters). The performance

of Regex, in general, is highly affected by the size of the string. The analysis also shows how various Regex features are being used within SPARQL queries. The major finding is that Regex patterns mainly consist of literal strings (not meta-characters), and that these literals tend to be long in their length.

Nevertheless, the actual performance of Regex within SPARQL has still not been adequately investigated. Most current benchmarks do not include Regex queries, nor has enough research been being done in this area. This thesis proposes an extension to a known benchmark, (BSBM), to include a set of queries to evaluate its performance, named BSBMstr. BSBMstr also added some other string matching functions that have been lately standardised by SPARQL 1.1, such as XPath functions, including STRSTARTS, STRENDS and CONTAINS. The results of running the benchmark has been reported against seven known RDF stores. The results also indicates the Regex features which mainly affect the performance of Regex within SPARQL, such as the cardinality of a given triple.

Studying benchmarks for the benefit of assessing Regex has shown a wider problem within SPARQL in general. Synthetic benchmarks, such as LUBM, are criticised in that they do not simulate real-world scenarios. Benchmarks with real data and queries usually tend to extract a smaller set of the queries to be representative and, therefore, used within the benchmark. We argue that this approach is not optimal for real benchmarks. The thesis, then, proposes a generic SPARQL benchmark, named CBSBench, which, unlike other real benchmarks, is based on clusters rather than queries. Each cluster has a set of queries that have been clustered, based on SPARQL features in common. In addition making a stress test on RDF stores (i.e., challenging caching of query results, plans etc), also allows the testing of RDF stores capability to produce correct results, because of the variety of queries being executed. The benchmark also reports its results on five different RDF stores.

Finally, given the analysis of how people are using Regex within SPARQL, and how Regex actually performs on RDF stores alongside a more reliable benchmark, we propose a Regex Index within SPARQL, (Reg|Ind)ex. Previous work has introduced the usage of n-grams to index RDF documents to optimise Regex queries with SPARQL. However, this approach yields a very large index size in relation to the main database size. Moreover, it uses a selectivity measure which, in some cases, requires the database to do a full scan when a gram is being removed, due to the threshold of frequency. (Reg|Ind)ex proposes a tri-gram index which produces a relatively much smaller index while maintaining query speed up to an order of magnitude. (Reg|Ind)ex was incorporated within two different RDF stores: Jena and Sesame, and then evaluated using two Regex-specific benchmarks (BSBMstr and CBSBench).

This work would mainly benefit developers who work on enhancing/testing the performance of executing queries against a particular DBMS, specifically SPARQL and/or

Regex. The thesis discussed and proposed various ideas and techniques that can be used by developers in the process of investigating a DBSM performance. For example, a developer whose task is to investigate the performance of Geo-spatial queries against a SQL database may benefit from the thesis by following the steps taken in this thesis. First, analysing the log files from their SQL server that contains those Geo-spatial queries. Identify some patterns that is being frequently used within the log queries. Then, the developer may also adapt the BSBMstr with their own set of queries which also can be reasoned about from the log files they analysed. A set of features about Geo-spatial queries could also been constructed with the help of some specifications about Geo-spatial queries. Having a set of features and some log files, it is also possible to adapt CBSBench benchmark. These benchmark could be sufficient to understand various aspect about the performance of Geo-spatial queries within SQL. For example, a particular set of Geo-spatial queries are shown to be slow in performance. Slow Geo-spatial queries could be mitigated by adopting a special index that, for example, reduce the number of data to be scanned. Some techniques we have used in this thesis is to reuse some other components with minor modifications such as: reusing the full-text search index (Lucene) to act as Regex index. In the case of Geo-spatial queries, Lucene also has a special index for these type of queries[1]. In general, in this thesis we provided a general framework that can be followed to investigate various performance aspects in similar areas.

[1] `https://lucene.apache.org/solr/guide/6_6/spatial-search.html`

7.1 Future Work

This thesis provides a great opportunity for further future work on various levels and topics; these include analysing query log files, benchmarking and optimising Regex performance. We discuss these possibilities below, based on the above topics.

7.1.1 Query Log Files Analysis

Chapter 3 presented a method to analyse the usage of Regex within SPARQL. Although different studies were proposed to analyse these log files, such as (Arias et al., 2011), there is little research being done in a more specific manner. It would be interesting to do similar research on the Property Paths introduced by SPARQL 1.1. We also discussed the issue of SPARQL standardisation throughout out this thesis: there are couple of extensions presented by RDF store vendors that are not part of the standardisation. A study of the state of SPARQL standardisation may find it useful to analyse the usage of these unsupported extensions.

7.1.2 SPARQL Benchmarking

Benchmarking has taken up quite a large proportion of this thesis's focus. Both Chapters 3 and 5 were about the techniques and methods for building a better tool to evaluate RDF stores (benchmarks). The area of real-world data and queries appears to have had more attention paid to it lately. Although CBSBench uses the K-means clustering algorithm as the standard choice, there are various algorithms which may show a better clustering of SPARQL queries. Moreover, the list of SPARQL features in which clustering is done based on them, was chosen based on their relevance as identified in the literature. We strongly encourage a study that presents a method to test a much larger proportion of SPARQL features, and probably rank them according to their general influence on the overall performance of SPARQL queries.

7.1.3 Regular Expressions Optimisations within SPARQL

This thesis has presented a proposal that optimises the performance of Regular Expressions within SPARQL, by means of a data structures approach. Another approach, that has not been investigated, is the optimisation of the computational side. Chapter 3 gave an indication that there are various patterns in which other algorithms can be used more efficiently than by using the generic Regex evaluator. In other words, the default behaviour of RDF stores is that, despite the Regex pattern being used within SPARQL, it will always be evaluated by a Regex engine attached to the RDF store. For example,

Regex patterns, such as "keyword" which does not contain Regex meta data could have been processed more efficiently by using a simpler string matching algorithm.

7.1.4 Automated Hybrid Approach

In the existing proposal (Reg|Ind)ex, one would have to explicitly state the use of the index by incorporating a special keyword within SPARQL, in which the query optimiser would trigger the index instead of the original implementation for Regex evaluator. In this thesis, we demonstrated that such approach would be beneficial to answer various Regex queries. However, The special keyword is not part of the SPARQL specifications. A further implementation may target using the default Regex syntax to be used within the scope of SPARQL specifications, then the query optimiser should then include some logic to decide whether the SPARQL with Regex pattern satisfies our index. For example, if it satisfies tri-gram sequences or having the evaluated string as Literal and not URI.

In addition to the above, we have also seen that some Regex queries performed better with the original implementation than our approach (See Figure 6.11 Query 3). This could be the result of various reasons such as: there is a large number of tri-grams to be scanned or joined to a point where a default Regex evaluator would have been more efficient. It is worth investigating the ability to adopt some heuristics through the usage of statistics about our index as well as about the original data within the query optimiser. These statistics should help the query optimiser to decide whether the Regex query should be evaluated with the Regex index or the original Regex evaluator. This practice of using statistics with query optimisers in the field of databases has been known and effective in many cases. Some approaches have already been discussed in Section 2.3.3.2.

7.1.5 Approximate Matching

Regular expressions can be seen as part of the exact matching problems. This means that for a given Regex pattern, the number of matched strings should be identical regardless of the Regex evaluator. In contrast to that, there is another class of problems that is under the category of approximate matching. This type considers finding related strings to the given pattern even if the returned string do not exactly satisfies the pattern. In SPARQL, those have been mainly discussed under Full-Text Search (FTS) extensions. FTS is still not considered within the SPARQL specification. Yet, most RDF stores implement their own extensions. We have discussed some implementations in Section 3.2. Moreover, in our proposal (Reg|Ind)ex, we have reused an existing FTS index to act as Regex index. In Section 3.4.4, we showed that there exists some FTS queries within the log files. Yet, there are still no published study that investigate how people

are using these queries within SPARQL, especially that they are not being standardised and how this might affect SPARQL queries.

7.1.6 Generalising to SQL

The thesis investigation is around the performance of SPARQL within Regex. A main motivation towards this choice was the coming reports about the performance of SPARQL not being as efficient as an equivalent SQL one, mainly caused by more generated joins within SPARQL. In addition to the known performance issues with Regex as well, this was a main driver towards this study. Having gone through this study, we think that it is still worth investigating the area of SQL performance within Regex queries. Unlike SPARQL where Regex is the only standardised way to perform a various string matching, SQL includes a simple string matching capability named LIKE which covers a broad use cases such as: contains, starts-with and ends-with. This will decrease the opportunity of gaining performance enhancement as in the case of SPARQL. Yet, This method would still require a full scanning of strings, in contrast to incorporating a Regex index. The later cause has a greater influence on the speed of queries as it known to run in order of millisecond instead of nanosecond for the complexity of the pattern itself. Another important factor to check before considering such approach is to consider the expected size of the Regex index and whether a DBMS would appreciate an increase in the size of the database that could well go over 100% depending on the data stored in the DBMS. Perhaps an analysis of queries stored in the log files or some use cases can help developers decide whether if such approach is worth investigating.

Appendix A

Regex Usage outputs

A.1 Predicates

```
1024422 http://www.w3.org/1999/02/22-rdf-syntax-ns#type
123684 http://dbpedia.org/property/name
87153 http://www.w3.org/2000/01/rdf-schema#label
69257 http://www.w3.org/2000/01/rdf-schema#subClassOf
39249 http://linkedgeodata.org/ontology/directType
29559 http://xmlns.com/foaf/0.1/name
23822 http://dbpedia.org/property/reference
17097 http://linkedgeodata.org/property/is_in
12959 http://linkedgeodata.org/property/is_in%3Acountry
12515 http://xmlns.com/foaf/0.1/page
10524 http://purl.org/dc/elements/1.1/title
9662 http://dbpedia.org/ontology/abstract
7883 http://purl.bioontology.org/ontology/NDFRT/NDFRT_KIND
5279 http://linkedgeodata.org/ontology/hasCity
4730 http://www.w3.org/2004/02/skos/core#subject
4639 http://www.w3.org/2002/07/owl#sameAs
2724 http://data.semanticweb.org/ns/swc/ontology#hasAcronym
2459 http://dbpedia.org/property/wikiPageUsesTemplate
2139 http://dbpedia.org/property/abstract
1973 http://dbpedia.org/property/leaderName
1266 http://xmlns.com/foaf/0.1/made
1238 http://www.w3.org/2000/01/rdf-schema#comment
1157 http://linkedgeodata.org/property/place
 892 http://www.w3.org/2000/01/rdf-schema#domain
 884 http://www.w3.org/2000/01/rdf-schema#range
 643 http://dbpedia.org/property/title
 458 http://www.w3.org/2003/01/geo/wgs84_pos#long
 458 http://www.w3.org/2003/01/geo/wgs84_pos#lat
 452 http://xmlns.com/foaf/0.1/firstName
 404 http://xmlns.com/foaf/0.1/surname
 390 http://dbpedia.org/property/influencedby
 390 http://dbpedia.org/property/influence
 390 http://dbpedia.org/property/,Influenced
 389 http://dbpedia.org/property/influenced
 388 http://dbpedia.org/property/influences
 388 http://dbpedia.org/property/influencedBy
 388 http://dbpedia.org/ontology/influencedBy
 388 http://dbpedia.org/ontology/influenced
 335 http://linkedgeodata.org/ontology/cuisine
 296 http://www.w3.org/2004/02/skos/core#prefLabel
 272 http://dbpedia.org/property/seat
 190 http://purl.org/net/chado/schema/annotationSymbol
 163 http://dbpedia.org/property/homepage
 160 http://xmlns.com/foaf/0.1/givenName
...
...
...
```

A.2 Regex Patterns and their occurrences

```
617226 ".+/((Company)|(Business)|(Company.*)|(CompaniesBasedIn.*)|(.*CompaniesOf.*))$"
199956 ".+/((Place)|(PopulatedPlace)|(Town)|(City)|(.*CitiesIn.*))$"
136695 ".+/((Place)|(PopulatedPlace))$"
127326 'http://dbpedia.org/resource/'
81390 "http://dbpedia.org/ontology/"
51871 "http://dbpedia.org/*"
23846 '^http://dbpedia.org/resource/Category:'
23842 '^http://sw.opencyc.org/'
23030 "http://www.w3.org/1999/02/22-rdf-syntax-ns#type*|http://dbpedia.org/ontology/type*"
20995 "(type|class|subject|broader)"
13632 '^http://dbpedia.org/resource/List'
13625 'yago'
13397 "freebase"
12558 'ontology'
11464 'twitter'
11464 'facebook'
11464 'en\\.wikipedia\\.org'
10754 "http://dbpedia.org/*|http://purl.org/dc/terms/subject|http://www.w3.org/1999/02/
22-rdf-syntax-ns#type*
|http://dbpedia.org/ontology/type*"
8298 "United States"
6390 "(sameas|type|class|subject|broader|label|name)"
5811 "http://dbpedia.org/resource/"
4243 "^http://dbpedia.org/"
3920 "ontology"
3619 "(label|summary|name)$"
3261 "^http://www.w3.org/2004/02/skos/core#subject"
3261 "^http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
3261 "^http://dbpedia.org/property/redirect"
3261 "^http://dbpedia.org/property/disambiguates"
3123 "http://www.w3.org/2000/01/rdf-schema#subClassOf"
2714 "Greatest Hits"
2585 "http://dbpedia.org/resource/Category:"
2448 "Top 1000 Pop Hits of the 80\'s"
2205 "http://dbpedia.org/class/yago/.*((MusicGroups)|(MusicalGroups))"
2062 "http://dbpedia.org/class/yago/.*((TelevisionSeries)|(TelevisionProgrammes)|(Telenovelas)|
(TelevisionSoapOperas)|(NetworkShows))"
2058 "http://dbpedia.org/resource/Category.*((TV)|(Television)).*series"
2039 "Spain"
1977 '^es$'
1961 "France"
1958 'http://rdf\\.freebase\\.com/.*'
1913 "^http://dbpedia.org/resource/"
1906 "pes"
1906 "Unknown"
1801 "^D"
1723 'http://data\\.nytimes\\.com/.*'
1676 "^http://data\\.semanticweb\\.org"
1575 "^B"
1557 "^http://dbpedia.org/ontology/"
1555 "Germany"
1368 "United Kingdom"
1341 "Italy"
1331 '[^a-z A-Z 0-9]*'
1308 "yago"
1296 "Australia"
1261 "http://dbpedia"
1153 "Top 1000 Pop Hits of the 80's"
1135 "http://purl.org/dc/terms/subject"
1058 "[?non-album tracks]?"
1037 "http://dbpedia.org/resource/Template:((infobox_disease)|(diseasedisorder_infobox))"
1037 "http://dbpedia.org/resource/Category:.*((diseases)|(disorders)|(pain)|
(abuse)|(conditions)|(symptoms)|(pathology)).*"
1037 "http://dbpedia.org/class/yago/.*((Disease)|(Disorder)|(Symptoms)).*"
```

Appendix B

BSBMstr queries and results

B.1 Regex Queries

Query1:

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

SELECT ?product ?label
WHERE {
 ?product rdfs:label ?label .
        FILTER regex (?label , "%word1%")
}
```

Query2:

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

SELECT ?product ?comment
WHERE {
 ?product rdfs:comment ?comment .
        FILTER regex (?comment , "%word1%")
}
```

Query3:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?reviewer ?reviewerName
WHERE {
 ?reviewer foaf:name ?reviewerName .
        FILTER regex (?reviewerName , "%word1%")
}
```

Query4:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?review ?title
WHERE {
 ?review dc:title ?title .
        FILTER regex (?title , "%word1%")
}
```

Query5:

```
PREFIX rev: <http://purl.org/stuff/rev#>

SELECT ?review ?text
WHERE {
 ?review rev:text ?text .
        FILTER regex (?text , "%word1%")
}
```

Query6:

```
PREFIX rev: <http://purl.org/stuff/rev#>

SELECT ?review ?text
WHERE {
 ?review rev:text ?text .
        FILTER regex (?text , "%word1%|%word2%|%word3%")
}
```

Query7:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?review ?title
WHERE {
 ?review dc:title ?title .
        FILTER regex (?title , ".*%word1%.*")
}
```

Query8:

```
PREFIX rev: <http://purl.org/stuff/rev#>

SELECT ?review ?text
WHERE {
 ?review rev:text ?text .
        FILTER regex (?text , "^%word1%")
}
```

Query9:

```
PREFIX rev: <http://purl.org/stuff/rev#>

SELECT ?review ?text
WHERE {
 ?review rev:text ?text .
        FILTER regex (?text , "%word1%$")
}
```

Query10:

```
PREFIX rev: <http://purl.org/stuff/rev#>

SELECT ?review ?text
WHERE {
 ?review rev:text ?text .
        FILTER regex (?text , "(\b[a-zA-Z]+\b).*\\1.*\\1")
}
```

B.2 XPath Queries

Query1:

```
PREFIX rev: <http://purl.org/stuff/rev#>

SELECT ?review ?text
WHERE {
 ?review rev:text ?text .
        FILTER STRSTARTS (?text , "%word1%")
}
```

Query2:

```
PREFIX rev: <http://purl.org/stuff/rev#>

SELECT ?review ?text
WHERE {
 ?review rev:text ?text .
        FILTER STRENDS (?text , "%word1%")
}
```

Query3:

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

SELECT ?product ?label
WHERE {
 ?product rdfs:label ?label .
        FILTER CONTAINS (?label , "%word1%")
}
```

Query4:

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

SELECT ?product
WHERE {
 ?product rdfs:label "%word1%" .
}
```

B.3 Regex XML Example Result

This is an example of the result produced by running the BSBM test driver on a Jena for 8M triples on single client:

```
<?xml version="1.0"?>
<bdsm version="Openlink BIBM Test Driver 0.7.7">
  <querymix>
     <date>Fri May 09 13:17:25 BST 2014</date>
     <scalefactor>0.0792764705469304</scalefactor>
     <exploreEndpoint>http://localhost:3030/BSBM/sparql</exploreEndpoint>
     <updateEndpoint>http://localhost:3030/BSBM/sparql</updateEndpoint>
     <usecase>/home/sza1g10/bibm8M/./queries/regex/BSBM_Filtered</usecase>
     <drilldown>off</drilldown>
     <warmups>50</warmups>
     <seed>808080</seed>
     <querymixruns>155</querymixruns>
     <minquerymixruntime>0.0000</minquerymixruntime>
     <maxquerymixruntime>176.0740</maxquerymixruntime>
     <elapsedruntime>17293.499</elapsedruntime>
     <qmph>32.266</qmph>
     <cqet>110.423</cqet>
     <cqetg>0.000</cqetg>
     <aqetg>2.656</aqetg>
     <throughput>17.411</throughput>
     <power>107.466</power>
  </querymix>
  <queries>
    <query nr="1">
      <executecount>106</executecount>
      <timeshare>0.172</timeshare>
      <aqet>0.277745</aqet>
      <aqetg>0.274127</aqetg>
      <qps>3.600</qps>
      <minqet>0.262000</minqet>
      <maxqet>0.722000</maxqet>
      <avgresults>2.368</avgresults>
      <minresults>1</minresults>
      <maxresults>14</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
    <query nr="2">
      <executecount>106</executecount>
      <timeshare>1.410</timeshare>
      <aqet>2.276792</aqet>
      <aqetg>2.272096</aqetg>
      <qps>0.439</qps>
      <minqet>2.167000</minqet>
      <maxqet>2.749000</maxqet>
      <avgresults>6.443</avgresults>
      <minresults>0</minresults>
      <maxresults>348</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
    <query nr="3">
      <executecount>106</executecount>
      <timeshare>0.048</timeshare>
      <aqet>0.076698</aqet>
      <aqetg>0.075513</aqetg>
      <qps>13.038</qps>
      <minqet>0.071000</minqet>
      <maxqet>0.174000</maxqet>
      <avgresults>0.000</avgresults>
      <minresults>0</minresults>
      <maxresults>0</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
    <query nr="4">
      <executecount>106</executecount>
      <timeshare>1.694</timeshare>
      <aqet>2.735972</aqet>
      <aqetg>2.735103</aqetg>
      <qps>0.366</qps>
      <minqet>2.615000</minqet>
      <maxqet>3.166000</maxqet>
      <avgresults>73.811</avgresults>
      <minresults>0</minresults>
      <maxresults>5190</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
```

```
    <query nr="5">
      <executecount>106</executecount>
      <timeshare>12.703</timeshare>
      <aqet>20.510736</aqet>
      <aqetg>20.493889</aqetg>
      <qps>0.049</qps>
      <minqet>14.624000</minqet>
      <maxqet>21.190000</maxqet>
      <avgresults>99.434</avgresults>
      <minresults>0</minresults>
      <maxresults>2741</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
    <query nr="6">
      <executecount>105</executecount>
      <timeshare>12.274</timeshare>
      <aqet>20.008010</aqet>
      <aqetg>19.257779</aqetg>
      <qps>0.050</qps>
      <minqet>0.913000</minqet>
      <maxqet>28.983000</maxqet>
      <avgresults>944.248</avgresults>
      <minresults>0</minresults>
      <maxresults>44170</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
    <query nr="7">
      <executecount>105</executecount>
      <timeshare>64.943</timeshare>
      <aqet>105.860895</aqet>
      <aqetg>105.835190</aqetg>
      <qps>0.009</qps>
      <minqet>102.039000</minqet>
      <maxqet>113.110000</maxqet>
      <avgresults>8.381</avgresults>
      <minresults>0</minresults>
      <maxresults>315</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
    <query nr="8">
      <executecount>105</executecount>
      <timeshare>0.165</timeshare>
      <aqet>0.268838</aqet>
      <aqetg>0.263273</aqetg>
      <qps>3.720</qps>
      <minqet>0.234000</minqet>
      <maxqet>0.700000</maxqet>
      <avgresults>1.019</avgresults>
      <minresults>0</minresults>
      <maxresults>74</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
    <query nr="9">
      <executecount>105</executecount>
      <timeshare>0.811</timeshare>
      <aqet>1.322333</aqet>
      <aqetg>1.276023</aqetg>
      <qps>0.756</qps>
      <minqet>0.933000</minqet>
      <maxqet>6.488000</maxqet>
      <avgresults>1.448</avgresults>
      <minresults>0</minresults>
      <maxresults>105</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
    <query nr="10"><executecount>105</executecount>
      <timeshare>5.932</timeshare>
      <aqet>9.669667</aqet><aqetg>9.666491</aqetg>
      <qps>0.103</qps><minqet>9.559000</minqet>
      <maxqet>11.565000</maxqet><avgresults>0.000</avgresults>
      <minresults>0</minresults><maxresults>0</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
  </queries>
</bdsm>
```

B.4 Jena Terminal Snapshot

This is a snapshot of running the BSBM test driver on Jena for testing regex queries end-point for 8M triples on a single client (Note: Those are not the data corresponding to the data being analysed; the below results are being only run for the purpose of showing an example):

```
[sza1g10@sociam-pub bibm8M]$ ./bsbmdriver -uc queries/regex/BSBM/ -w 50 -o
Jena/Regex_1Client_BSBM8M_.xml -runs 200 http://localhost:3030/BSBM/sparql
java -Xmx256M com.openlinksw.bibm.bsbm.TestDriver -qrd . -uc queries/regex/BSBM/
-w 50 -o Jena/Regex_1Client_BSBM8M_.xml -runs 50 http://localhost:3030/BSBM/sparql

Reading Test Driver data...
Reading query ignore file: ./queries/regex/BSBM/ignoreQueries.txt
Reading query mix file: ./queries/regex/BSBM/querymix.txt

Starting test...

-- preparation time=871
Thread 1: query mix: -50  0.000 s, total: 166.736 s
Thread 1: query mix: -49  0.000 s, total: 174.278 s
Thread 1: query mix: -48  0.000 s, total: 172.159 s
Thread 1: query mix: -47  0.000 s, total: 173.685 s
Thread 1: query mix: -46  0.000 s, total: 173.030 s
Thread 1: query mix: -45  0.000 s, total: 172.229 s
Thread 1: query mix: -44  0.000 s, total: 169.895 s
Thread 1: query mix: -43  0.000 s, total: 170.157 s
Thread 1: query mix: -42  0.000 s, total: 170.540 s
Thread 1: query mix: -41  0.000 s, total: 180.471 s
...
...
...
Warmup phase ended...

Starting actual run...
Thread 1: query mix: 0  171.188 s, total: 171.209 s
Thread 1: query mix: 1  171.148 s, total: 171.188 s
Thread 1: query mix: 2  171.951 s, total: 171.969 s
Thread 1: query mix: 3  144.057 s, total: 170.575 s
Thread 1: query mix: 4  157.233 s, total: 173.423 s
Thread 1: query mix: 5  177.724 s, total: 177.743 s
Thread 1: query mix: 6  168.988 s, total: 169.011 s
Thread 1: query mix: 7  171.325 s, total: 171.345 s
Thread 1: query mix: 8  172.971 s, total: 173.018 s
Thread 1: query mix: 9  168.798 s, total: 168.817 s
Thread 1: query mix: 10  171.344 s, total: 171.362 s
Thread 1: query mix: 11  172.017 s, total: 172.037 s
Thread 1: query mix: 12  171.156 s, total: 171.196 s
Thread 1: query mix: 13  172.265 s, total: 172.284 s
Thread 1: query mix: 14  156.388 s, total: 173.285 s
Thread 1: query mix: 15  163.314 s, total: 166.996 s
Thread 1: query mix: 16  164.584 s, total: 171.886 s
Thread 1: query mix: 17  172.792 s, total: 172.812 s
Thread 1: query mix: 18  169.475 s, total: 169.491 s
...
...
...
```

Appendix C

CBSBench Data and Results Snippets

C.1 Query Cluster 1 snippet

```
SELECT * WHERE { <http://dbpedia.org/resource/NotALink> dbpprop:redirect ?var0 } OFFSET 0 LIMIT 10
SELECT ?var0 WHERE { <http://dbpedia.org/resource/> <http://dbpedia.org/ontology/abstract> ?var0
FILTER langMatches(lang(?var0), "en") } OFFSET 0 LIMIT 10
SELECT * WHERE { <http://dbpedia.org/resource/NotALink> rdfs:label ?var0 } OFFSET 0 LIMIT 10
SELECT ?var0 WHERE { <http://dbpedia.org/ontology/Film>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?var0 }
OFFSET 0 LIMIT 10
SELECT ?var0 WHERE { <http://dbpedia.org/resource/Alaskan_Malamute>
<http://dbpedia.org/property/color> ?var0 } OFFSET 0 LIMIT 10
SELECT * WHERE { <http://dbpedia.org/resource/NotALink> ?var0
<http://dbpedia.org/resource/NotALink> } OFFSET 0 LIMIT 10
SELECT ?var0 WHERE { <http://dbpedia.org/resource/Jean_Reno>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?var0
} OFFSET 0 LIMIT 10
SELECT ?var0 WHERE { <http://dbpedia.org/resource/Stuttgart>
<http://dbpedia.org/property/aprHi_percent_C2_percent_B0c> ?var0
} OFFSET 0 LIMIT 10
SELECT ?var0 WHERE { <http://dbpedia.org/resource/Stuttgart>
<http://dbpedia.org/property/locode> ?var0 } OFFSET 0 LIMIT 10
SELECT ?var0 WHERE { <http://dbpedia.org/resource/St._Petersburg%2C_Florida>
<http://dbpedia.org/property/imageMap> ?var0 } OFFSET 0 LIMIT 10
SELECT ?var0 WHERE { <http://dbpedia.org/resource/The_Quarrymen>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?var0 } OFFSET 0 LIMIT 10
...
...
298,744 lines/queries
```

C.2 Weka clustering Outputs

This is the outputs of running Weka clustering algorithm. The output, below, is displayed within four pages start from this point.

```
=== Run information ===

Scheme:weka.clusterers.SimpleKMeans -N 25 -A "weka.core.EuclideanDistance -R first-last"
 -I 1000 -O -S 10
Relation:     ClustersAddedBack-weka.filters.unsupervised.attribute.Remove-R33-
weka.filters.unsupervised.attribute.Remove-R13-weka.filters.unsupervised.attribute.
AddCluster-Wweka.clusterers.SimpleKMeans -N 25 -A "weka.core.EuclideanDistance -R
 first-last" -I 1000 -O -S 10
Instances:    1280930
Attributes:   32
               select
               construct
               describe
               ask
               order by
               limit
               offset
               distinct
               optional
               union
               filter
               lang
               bound
               regex
              SubSub
               PrePre
               ObjObj
               SubPre
               SubObj
               PreObj
              VVV
               VVC
               VCC
               CVV
               CCC
               CVC
               VCV
               CCV
               triple patterns
              StarOutdegree
              LongOfChain
              cluster
Test mode:evaluate on training data

=== Model and evaluation on training set ===
```

```
Number of iterations: 5
Within cluster sum of squared errors: 21168.280304572225
Missing values globally replaced with mean/mode

Cluster centroids:
                          Cluster#
Attribute       Full Data         0         1         2         3         4         5         6         7         8         9        10        11        12
                (1280930)  (298744)   (8966)  (10940)  (237523)  (23467)   (6987)  (13862)  (150153)  (16738)  (119656)  (16176)   (9223)  (46316)
=============================================================================================================================================================
 select            0.9622         1         0         1         1         1         1         1         1         0         1    0.9998         1         1
 construct         0.0063         0    0.9069         0         0         0         0         0         0         0         0         0         0         0
 describe          0.0007         0    0.0931         0         0         0         0         0         0         0         0    0.0002         0         0
 ask               0.0308         0         0         0         0         0         0         0         0         1         0         0         0         0
 order by          0.0025         0    0.0001         0         0         0    0.0003         0         0         0         0    0.0192    0.0024    0.0173
 limit             0.0283         0    0.0622         0         0         0         0         0         0         0         0         1         0         0
 offset            0.0014         0         0         0         0         0         0         0         0         0         0         0         0         0
 distinct          0.2308         0         0         0         0    0.0017         0         0         1         0         0    0.1101         0         0
 optional          1.2984    0.0002     1.492    0.0003    0.9992    1.9974         0    0.0004    0.0357         0    2.9992    0.3929    0.7235     0.023
 union             0.8757    0.0005    0.0293    0.0004    0.0001    0.0133    0.5057    0.0001    0.0152         0    3.9117    0.2708    1.4789    0.2576
 filter            0.4405    0.3322    0.9051         0         0     1.022    0.4842    1.0003    0.1798    0.0009    1.0004    1.3795    1.1246     0.046
 lang              0.5012    0.3181    1.1604         0         0    5.9987         0    0.0019    0.1789         0    0.9996    0.6525     1.117    0.0206
 bound             0.0934         0         0         0         0         0         0         0         0         0         1    0.0004         0         0
 regex             0.0241    0.0073    0.0477         0         0         0    0.2665    0.1337    0.0003         0         0    0.1098         0    0.0143
SubSub             2.9741    0.0421    0.9772    0.0017     1.998    0.0006    0.3156         0    0.0489         0    9.8803    2.3384    2.3966    1.5139
 PrePre            0.5353    0.0001      0.58    0.0004         0    1.0021    0.7733    0.0002    0.0103    0.0004    1.9414    0.1013    0.0223    0.2645
 ObjObj            0.6783         0    0.0445    0.0001    0.0001         0         0    0.0014    0.0058         0    2.9414    0.2583    0.0378    0.0025
 SubPre            0.0195         0    0.0084    0.0005         0    0.9987         0    0.0002    0.0076         0         0    0.0004         0    0.0001
 SubObj            0.6197    0.0002    0.6734    0.0005    0.0001     0.999    0.7992    0.0001    0.0453         0    3.8828    0.0691    0.0381    0.0133
 PreObj                 0         0         0         0         0         0         0         0         0         0         0    0.0109         0         0
VVV                0.0015    0.0018    0.1006    0.0002         0         0    0.0006         0    0.0002    0.0001         0    0.2357         0    0.0009
 VVC               0.0252         0    0.0016         0         0         0    0.1029         0         0    0.0001         0    0.0001         0         0
 VCC               0.8803         0    0.0435    0.0003         0    0.0003    0.0014         0    0.0089         0    2.9988    0.7604     0.019    1.0611
 CVV               0.0706         0      0.12    1.0003         0    0.9987    0.5108    1.0018         0    0.9314         0    0.0047         0         0
 CCC               0.0111    0.0416         0    0.0006         0         0         0         0    0.0076    0.0212         0    0.0017         0    0.0002
 CVC               0.0029    0.0123         0         0         0         0         0         0         0    0.0005         0    0.0001         0         0
 VCV               2.0558     0.005    1.1118    0.0014    0.0001    2.0024    0.6004    0.0003    0.1939    0.0002    8.8522    2.3605    0.0642     1.465
 CCV                1.206     0.982    1.2231    0.0004    2.9981         0         0         0    0.8834     0.047    0.0004    0.0386    3.4469    0.0016
 triple patterns   4.2534    1.0427    2.6005    1.0031    2.9981    3.0014    2.1161    1.0021     1.094    1.0004    11.8514    3.4126    3.5301    2.5288
StarOutdegree      2.5555    0.0421    0.9568    0.0012     1.002    0.0005     0.294         0    0.0486         0    9.8803    1.4463    2.3746    1.4719
LongOfChain        1.1387    1.0001    1.6566    1.0004         1    1.9988    1.2967    1.0001    1.0361         1         1    1.0483    1.0086    1.0069
cluster            cluster1  cluster1  cluster2  cluster3  cluster4  cluster5  cluster6  cluster7  cluster8  cluster9  cluster10 cluster11 cluster12 cluster13
```

Cluster centroids:

Attribute	Full Data (1280930)	Cluster# 13 (105640)	14 (16138)	15 (12434)	16 (22736)	17 (19560)	18 (30892)	19 (47988)	20 (20266)	21 (20541)	22 (24697)	23 (51)	24 (1236)
select	0.9622	1	1	1	0	1	1	1	1	1	1	1	1
construct	0.0063	0	0	0	0	0	0	0	0	0	0	0	0
describe	0.0007	0	0	0	0	0	0	0	0	0	0	0	0
ask	0.0308	0	0	0	1	0	0	0	0	0	0	0	0
order by	0.0025	0	0	0.0037	0	0.0937	0	0.0036	0.0003	0	0	0	0
limit	0.0283	0	0	0	0	1	0	0	0	0	0	0	0
offset	0.0014	0	0	0	0	0	0	0.0007	0	0	0	0	0
distinct	0.2308	0.0005	0.0014	0.0066	0	0.9921	0	1	1	1	0.9578	1	0
optional	1.2984	6.0081	4.523	1.7058	0	11.6074	0	0.1152	0	0.0001	1.0395	0	0
union	0.8757	2.0035	0.9899	2.4571	0	4.2496	0.0001	0.0342	0.0002	0.0021	11.0492	0	1
filter	0.4405	1.0087	1.0128	0.0791	0.0001	2.8905	0.0025	0.1338	0.0002	0.0019	1.8082	1	2
lang	0.5012	0.9983	2.0034	0.0104	0	2.8905	0.0001	0.1289	0.0001	0.0002	0.9839	0	2.0243
bound	0.0934	0	0	0	0	0	0	0	0	0	0	0	0
regex	0.0241	0.0097	0.0014	0.0014	0	0	0.0023	0.0047	0.0002	0.0003	0.8236	8	0
SubSub	2.9741	10.0168	9.9671	6.9579	0.9994	16.8598	0.0119	0.3184	0.0016	0.0052	12.8763	0	0
PrePre	0.5353	2.0055	2.0059	1.6261	0	0.1016	0.0004	0.0613	0.0003	0	5.3812	0	1
ObjObj	0.6783	3.0028	0.9894	1.7036	0	4.2496	0.0001	0.0086	0	0.0047	2.956	0	0
SubPre	0.0195	0	0	0	0	0	0	0.0068	0	0	0	0	0
SubObj	0.6197	2.0029	1.0058	1.7636	0	0	0.0019	0.0335	0.0006	0.0003	1.3177	0	1
PreObj	0	0	0	0.0001	0	0	0	0	0	0	0	0	0
VVV	0.0015	0.0001	0	0.0053	0	0	0.0008	0.0019	0	0	0	0	0
VVC	0.0252	0	0	0	0	0	0	0	1	0	0	0	1
VCC	0.8803	3.0001	2.9789	3.2812	1.9992	0.1016	1.0116	1.0505	0.0009	0.0003	6.9046	0	0
CVV	0.0706	0	0	0	0	0	0	0	0.0003	1.0021	0	1	1
CCC	0.0111	0	0	0	0	0	0	0.0043	0	0	0	0	0
CVC	0.0029	0	0	0	0	0	0	0	0	0	0	0	0
VCV	2.0558	9.0167	8.9576	5.5798	0.0002	1.5868	0.0014	0.2807	0.0009	0.0003	6.6659	0	0
CCV	1.206	0.003	0.0359	0.0117	0	16.1715	0.0005	0.0207	0	0.0026	1.778	0	0
triple patterns	4.2534	12.0199	11.9724	8.878	1.9994	17.8598	1.0144	1.3581	1.0022	1.0055	15.3485	1	2
StarOutdegree	2.5555	8.0162	7.4998	6.9463	0.9993	15.9614	0.0119	0.2637	0.001	0.0029	12.4276	0	0
LongOfChain	1.1387	1.9993	2.0035	1.0136	1	1	1.0007	1.0137	1.0003	1.0003	1.6887	1	1
cluster	cluster14	cluster15	cluster15	cluster16	cluster17	cluster18	cluster19	cluster20	cluster21	cluster22	cluster23	cluster24	cluster25

```
Time taken to build model (full training data) : 134.5 seconds

=== Model and evaluation on training set ===

Clustered Instances

 0       298744 ( 23%)
 1         8966 (  1%)
 2        10940 (  1%)
 3       237523 ( 19%)
 4        23467 (  2%)
 5         6987 (  1%)
 6        13862 (  1%)
 7       150153 ( 12%)
 8        16738 (  1%)
 9       119656 (  9%)
10        16176 (  1%)
11         9223 (  1%)
12        46316 (  4%)
13       105640 (  8%)
14        16138 (  1%)
15        12434 (  1%)
16        22736 (  2%)
17        19560 (  2%)
18        30892 (  2%)
19        47988 (  4%)
20        20266 (  2%)
21        20541 (  2%)
22        24697 (  2%)
23           51 (  0%)
24         1236 (  0%)
```

The Weka clustering output results ends here.

C.3 Weka Clustering Logging Procedure

```
17:42:48: Weka Explorer
17:42:48: (c) 1999-2014 The University of Waikato, Hamilton, New Zealand
17:42:48: web: http://www.cs.waikato.ac.nz/~ml/weka/
17:42:48: Started on Thursday, 2 June 2016
17:43:26: Command: weka.filters.unsupervised.attribute.Remove -R 1-2,8-9,11-15,19-27,30-38
17:43:50: Command: weka.filters.unsupervised.attribute.Remove -R 12,16-17,25
17:45:17: Command: weka.filters.unsupervised.attribute.AddCluster -W
"weka.clusterers.SimpleKMeans -N 15 -A \"weka.core.EuclideanDistance -R first-last\"
 -I 500 -O -S 10"
17:45:34: Started weka.clusterers.SimpleKMeans
17:45:34: Command: weka.clusterers.SimpleKMeans -N 15 -A "weka.core.EuclideanDistance -R
first-last" -I 500 -O -S 10
17:45:35: Finished weka.clusterers.SimpleKMeans
17:46:27: Command: weka.filters.unsupervised.attribute.Remove -R 24
17:46:46: Command: weka.filters.unsupervised.attribute.AddCluster -W
"weka.clusterers.SimpleKMeans -N 25 -A \"weka.core.EuclideanDistance -R first-last\"
 -I 500 -O -S 10"
17:46:55: Started weka.clusterers.SimpleKMeans
17:46:55: Command: weka.clusterers.SimpleKMeans -N 20 -A "weka.core.EuclideanDistance
 -R first-last" -I 500 -O -S 10
17:46:56: Finished weka.clusterers.SimpleKMeans
```

C.4 CBSBench Running Output for Virtuoso

```
java -Xmx5G com.openlinksw.bibm.bsbm.TestDriver -qrd . -uc queries/EuclideanLimit10/ -w 5 -runs 50
 -o DBpediaQueries3_5_1_100Percent/EuclideanLimit10/
 virtuoso7_DBpediaClusters_Generic_100Percent_single.xml http://localhost:8890/sparql
Reading Test Driver data...
Reading query ignore file: ./queries/EuclideanLimit10/ignoreQueries.txt
Reading query mix file: ./queries/EuclideanLimit10/querymix.txt

Starting test...

-- preparation time=4162
QueryID: 1
QueryCluster: 1
LineNo: 194614
Query String: SELECT ?var0 WHERE { <http://dbpedia.org/resource/Geology_of_Orkney> rdf:type ?var0 }
OFFSET 0 LIMIT 10
Time In Seconds: 0.034
Result Count: 0
QueryID: 2
QueryCluster: 2
LineNo: 3763
Query String: CONSTRUCT { <http://dbpedia.org/resource/Machete> rdfs:label ?var0 .
<http://dbpedia.org/resource/Machete> rdf:type ?var1 . ?var1 rdfs:label ?var3 .}
WHERE { <http://dbpedia.org/resource/Machete> rdfs:label ?var0 FILTER
langMatches(lang(?var0), "en") OPTIONAL { <http://dbpedia.org/resource/Machete>
rdf:type ?var1 } OPTIONAL { ?var1 rdfs:label ?var3 } }
OFFSET 0 LIMIT 10
Time In Seconds: 0.002
Result Count: 0
QueryID: 3
QueryCluster: 3
LineNo: 4703
Query String: SELECT ?var0 ?var1 WHERE { :Histopathology ?var0 ?var1 }
OFFSET 0 LIMIT 10
Time In Seconds: 0.003
Result Count: 10
...
...
QueryID: 369
QueryCluster: 19
LineNo: 21935
Query String: SELECT * WHERE { ?var0 foaf:page <http://en.wikipedia.org/wiki/Titanium_Metals> }
OFFSET 0 LIMIT 10
Time In Seconds: 0.002
Result Count: 1
QueryID: 370
QueryCluster: 20
LineNo: 40378
Query String: SELECT DISTINCT ?var0 WHERE { ?var0 dbpedia-prop:redirect
<http://dbpedia.org/resource/Raznyie_Lyudi> } OFFSET 0 LIMIT 10
Time In Seconds: 0.003
Result Count: 0
QueryID: 371
QueryCluster: 21
LineNo: 11351
Query String: SELECT DISTINCT ?var0 ?var1 WHERE { ?var0 ?var1
<http://dbpedia.org/resource/Category:1991_albums> } OFFSET 0 LIMIT 10
Time In Seconds: 0.033
Result Count: 10
QueryID: 372
QueryCluster: 22
LineNo: 13357
Query String: SELECT DISTINCT ?var0 ?var1 WHERE { <http://www.mietoinen.fi/>
 ?var0 ?var1 } OFFSET 0 LIMIT 10
Time In Seconds: 0.032
Result Count: 0
...
```

C.5 CBSBench Result Output for Virtuoso

```
<?xml version="1.0"?>
<bdsm version="Openlink BIBM Test Driver 0.7.7">
  <querymix>
     <date>Tue Aug 18 12:02:00 BST 2015</date>
     <scalefactor>0.009777899489512895</scalefactor>
     <exploreEndpoint>http://localhost:8890/sparql</exploreEndpoint>
     <updateEndpoint>http://localhost:8890/sparql</updateEndpoint>
     <usecase>/home/sza1g10/bibmDBpediaGeneric/./queries/EuclideanLimit10</usecase>
     <drilldown>off</drilldown>
     <warmups>5</warmups>
     <seed>808080</seed>
     <querymixruns>55</querymixruns>
     <minquerymixruntime>0.0000</minquerymixruntime>
     <maxquerymixruntime>3.2110</maxquerymixruntime>
     <elapsedruntime>73.929</elapsedruntime>
     <qmph>2678.246</qmph>
     <cqet>1.224</cqet>
     <cqetg>0.000</cqetg>
     <aqetg>0.015</aqetg>
     <throughput>595.173</throughput>
     <power>2406.480</power>
  </querymix>
  <queries>
    <query nr="1">
      <executecount>50</executecount>
      <timeshare>3.823</timeshare>
      <aqet>0.051480</aqet>
      <aqetg>0.008375</aqetg>
      <qps>19.425</qps>
      <minqet>0.001000</minqet>
      <maxqet>0.526000</maxqet>
      <avgresults>1.280</avgresults>
      <minresults>0</minresults>
      <maxresults>5</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
    <query nr="2">
      <executecount>50</executecount>
      <timeshare>4.276</timeshare>
      <aqet>0.057580</aqet>
      <aqetg>0.008366</aqetg>
      <qps>17.367</qps>
      <minqet>0.001000</minqet>
      <maxqet>1.245000</maxqet>
      <avgresults>0.000</avgresults>
      <minresults>0</minresults>
      <maxresults>0</maxresults>
      <timeoutcount>0</timeoutcount>
    </query>
...
...
...
```

Appendix D

(Reg|Ind)ex: Regular Expressions Indexing

D.1 Queries with BSBMstr (Jena and Sesame)

Query1 (Jena):

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

SELECT ?product ?label
WHERE {
 ?product text:regex (rdfs:label "%word1%") .
 ?product rdfs:label ?label .
        FILTER regex (?label , "%word1%")
}
```

Query2 (Sesame):

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

SELECT ?product ?comment
WHERE {
 ?regexIntermiedate ls:query "%word1%" .
 ?product ls:matches ?regexIntermiedate .
 ?product rdfs:comment ?comment .
        FILTER regex (?comment , "%word1%")
}
```

Query3 (Jena):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX text: <http://jena.apache.org/text#>

SELECT ?reviewer ?reviewerName
WHERE {
 ?product text:regex (foaf:name "%word1%") .
 ?product foaf:name ?reviewerName .
        FILTER regex (?reviewerName , "%word1%")
}
```

...

...

...

Query7 (Sesame):

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ls: <http://www.openrdf.org/contrib/lucenesail#>

SELECT ?review ?title
WHERE {
 ?regexIntermiedate ls:query ".*%word1%.*" .
 ?review ls:matches ?regexIntermiedate .
 ?review dc:title ?title .
        FILTER regex (?title , ".*%word1%.*")
}
```

Query8 (Jena):

```
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX text: <http://jena.apache.org/text#>

SELECT ?review ?text
WHERE {
 ?review text:regex (rev:text "^%word1%") .
 ?review rev:text ?text .
        FILTER regex (?text , "^%word1%")
}
```

Query9 (Sesame):

```
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX ls: <http://www.openrdf.org/contrib/lucenesail#>

SELECT ?review ?text
WHERE {
 ?regexIntermiedate ls:query "%word1%$" .
 ?review ls:matches ?regexIntermiedate .
 ?review rev:text ?text .
        FILTER regex (?text , "%word1%$")
}
```

Query10 (Both Jena and Sesame):

```
PREFIX rev: <http://purl.org/stuff/rev#>

SELECT ?review ?text
WHERE {
 ?review rev:text ?text .
        FILTER regex (?text , "(\b[a-zA-Z]+\b).*\\1.*\\1")
}
```

D.2 Queries with CBSBench (Sesame)

```
Cluster12:SELECT DISTINCT ?var0 ?var1 WHERE { ?regexIntermiedate
<http://www.openrdf.org/contrib/lucenesail#property> rdfs:label .
?regexIntermiedate <http://www.openrdf.org/contrib/lucenesail#query>
"Award" . ?var3 <http://www.openrdf.org/contrib/lucenesail#matches>
?regexIntermiedate . <http://dbpedia.org/resource/MirrorMask> foaf:name
?var2 OPTIONAL { ?regexIntermiedate <http://www.openrdf.org/contrib/
lucenesail#property> rdfs:label . ?regexIntermiedate
<http://www.openrdf.org/contrib/lucenesail#query> "Award" .
?var3 <http://www.openrdf.org/contrib/lucenesail#matches>
?regexIntermiedate . <http://dbpedia.org/resource/MirrorMask>
dcterms:subject ?var3 } OPTIONAL { ?regexIntermiedate
<http://www.openrdf.org/contrib/lucenesail#property> rdfs:label .
?regexIntermiedate <http://www.openrdf.org/contrib/lucenesail#query>
"Award" . ?var3 <http://www.openrdf.org/contrib/lucenesail#matches>
?regexIntermiedate . ?var3 rdfs:label ?var0
FILTER regex(str(?var0), "Award") } OPTIONAL { ?regexIntermiedate
<http://www.openrdf.org/contrib/lucenesail#property> rdfs:label .
?regexIntermiedate <http://www.openrdf.org/contrib/lucenesail#query>
"Award" . ?var3 <http://www.openrdf.org/contrib/lucenesail#matches>
?regexIntermiedate . ?var3 rdfs:label ?var1 FILTER regex(str(?var1),
"[0-9]{4} films") } } OFFSET 0 LIMIT 10
...
...
Cluster17:SELECT ?var0 WHERE { ?regexIntermiedate
<http://www.openrdf.org/contrib/lucenesail#property> dbpedia2:name .
 ?regexIntermiedate <http://www.openrdf.org/con
trib/lucenesail#query> "Horse Of A Different Color" .
?var1 <http://www.openrdf.org/contrib/lucenesail#matches>
?regexIntermiedate . ?var1 dbpedia2:name ?var2 . ?var1 dbpedia2:artist
?var4 .?var4 dbpedia2:name "Big & Rich"@en . ?var1 rdf:type owl:Album .
?var1 dbpedia2:cover ?var0 FILTER regex(str(?var2), "Horse Of
A Different Color"@en, "i") } OFFSET 0 LIMIT 10
...
...
Cluster15:SELECT DISTINCT ?var0 WHERE { GRAPH ?var1 {
?regexIntermiedate <http://www.openrdf.org/contrib/lucenesail#property>
rdfs:label . ?regexIntermiedate <http://www.openrdf.org/contrib/
lucenesail#query> "Cat". ?var2 <http://www.openrdf.org/contrib/
lucenesail#matches> ?regexIntermiedate . ?var2 rdfs:label ?var2earch .
?var2 rdf:type ?var0 FILTER regex(str(?var2earch), "Cat")
FILTER langMatches(lang(?var2earch), "en")
FILTER regex(str(?var0), "^http://dbpedia.org/") } } OFFSET 0 LIMIT 10
...
...
Cluster24:SELECT DISTINCT * WHERE { ?regexIntermiedate
<http://www.openrdf.org/contrib/lucenesail#property> rdfs:label .
?regexIntermiedate <http://www.openrdf.org/contrib/lucenesail#query>
"Japan" . ?var0 <http://www.openrdf.org/contrib/lucenesail#matches>
?regexIntermiedate . ?var0 rdfs:label "Japan"@en .
?var0 rdfs:label ?var2 . ?var0 rdf:type ?var4
FILTER regex(str(?var2), "Japan", "i") } OFFSET 0 LIMIT 10
...
...
Cluster24:SELECT DISTINCT ?var0 ?var1 WHERE { ?regexIntermiedate
<http://www.openrdf.org/contrib/lucenesail#property> rdfs:label .
?regexIntermiedate <http://www.openrdf.org/contrib/
lucenesail#query> "Hobart" . ?var0 <http://www.openrdf.org/contrib/
lucenesail#matches> ?regexIntermiedate . ?var0 rdf:type schema:City .
?var0 rdfs:label ?var1 FILTER regex(str(?var1), "Hobart") }
OFFSET 0 LIMIT 10
...
...
```

D.3 Source Code

D.3.1 Building the Index

```
                analyzerPerField.put(def.getEntityField(), new WhitespaceAnalyzer(VER));
                if (def.getGraphField() != null)
                        analyzerPerField.put(def.getGraphField(), new WhitespaceAnalyzer(
                                        VER));
                this.analyzer = new PerFieldAnalyzerWrapper(
                                new WhitespaceAnalyzer(VER), analyzerPerField);

        private Document doc(Entity entity) {
                 FieldType ftString = new FieldType(StringField.TYPE_NOT_STORED);


                 FieldType ftText = new FieldType(TextField.TYPE_NOT_STORED);

                Document doc = new Document();
                ftString.setIndexOptions(FieldInfo.IndexOptions.DOCS_ONLY);
                ftText.setIndexOptions(FieldInfo.IndexOptions.DOCS_ONLY);

                Field entField = new Field(docDef.getEntityField(), entity.getId(),
                                ftIRI);

                doc.add(entField);

                String graphField = docDef.getGraphField();
                if (graphField != null) {

                        Field gField = new Field(graphField, entity.getGraph(), ftString);

                        doc.add(gField);
                }
                for (Entry<String, Object> e : entity.getMap().entrySet()) {

                        String processedText = fromFileToHashSet((String) e.getValue());
                        Field field = new Field(e.getKey(), processedText, ftText);

                        doc.add(field);
                }
                return doc;
        }

        public String fromFileToHashSet(String line) {
                HashSet<String> ngramsList = new HashSet<String>();

                for (int i = 0; i <= line.length() - trigram; i++) {
                        String ngram = line.substring(i, trigram + i);
                        if (!ngram.contains(" ")) {
                                ngramsList.add(ngram.toLowerCase());
                        }
                }

                StringBuilder builder = new StringBuilder();
                for (String string : ngramsList) {
                        builder.append(string + " ");
                }

                return builder.toString();
        }
```

D.3.2 Querying the Index

```
	int trigram = 3;
	private String booleanQuerySyntax(String regex) {
		StringBuilder parsed = new StringBuilder();
		Pattern pattern2 = Pattern.compile("[^\\\\](" +
				"\\." +
				"|\\*" +
				"|\\+" +
				"|\\^" +
				"|\\$" +
				"|\\?" +
				"|\\[" +
				"|\\]" +
				"|\\(" +
				"|\\)" +
				"|\\{" +
				"|\\}" +
				"|\\\\" +
				")");

		if (regex.contains("|")) {
			String[] splitter = regex.split("\\|");
			parsed.append("(");
			for (String string : splitter) {
				parsed.append("(");
				for (int i = 0; i <= string.length() - trigram; i++) {
					String ngram = string.substring(i, trigram + i);
					Matcher matcher = pattern2.matcher(ngram);
					if (!matcher.find()) {
						parsed.append(ngram.toLowerCase());
						parsed.append(" AND ");
					}
				}
				parsed.delete(parsed.length() - 5, parsed.length());
				parsed.append(")");
				parsed.append(" OR ");
			}
			parsed.delete(parsed.length() - 4, parsed.length());
			parsed.append(")");

		} else {
			for (int i = 0; i <= regex.length() - trigram; i++) {
				String ngram = regex.substring(i, trigram + i);
				Matcher matcher = pattern2.matcher(ngram);
				if (!matcher.find()) {
					parsed.append(ngram.toLowerCase() + " AND ");
				}
			}
			parsed.delete(parsed.length() - 5, parsed.length());
		}
		return parsed.toString();

	}
```

```
	private List<Node> query$(IndexReader indexReader, String qs, int limit)
			throws ParseException, IOException {

		IndexSearcher indexSearcher = new IndexSearcher(indexReader);
		QueryParser queryParser = new QueryParser(VER,
				docDef.getPrimaryField(), analyzer);
		Query query;
		String booleanQuery;
		if (qs.contains(":")) {
			String searchFeild = qs.replaceFirst(":.*", "");
			String cutColon = qs.replaceFirst(".*?:", "");
			booleanQuery = booleanQuerySyntax(cutColon);
			String overAllQuery = searchFeild + ":" + "(" + booleanQuery + ")";
			query = queryParser.parse(overAllQuery);
		} else {
			booleanQuery = booleanQuerySyntax(qs);
			query = queryParser.parse(booleanQuery);

		}

		if (limit <= 0)
			limit = MAX_N;

		TotalHitCountCollector collector = new TotalHitCountCollector();
		indexSearcher.search(query, collector);
		TopDocs topDocs = indexSearcher.search(query, Math.max(1,
		collector.getTotalHits()));
		List<Node> results = new ArrayList<Node>();

		// Align and DRY with Solr.
		for (ScoreDoc sd : topDocs.scoreDocs) {
			Document doc = indexSearcher.doc(sd.doc);
			String[] values = doc.getValues(docDef.getEntityField());
			for (String v : values) {
				Node n = TextQueryFuncs.stringToNode(v);
				results.add(n);
			}
		}
		return results;
	}
```

References

(2002). *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, San Jose, CA, USA. IEEE Computer Society.

(2004). *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, New York, NY, USA. ACM.

(2011). *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, Bonn, Germany. Springer.

(2013). *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, volume 8219 of *Lecture Notes in Computer Science*, Sydney, NSW, Australia. Springer.

Abadi, D. J., Marcus, A., Madden, S., and Hollenbach, K. (2009). Sw-store: a vertically partitioned DBMS for semantic web data management. *VLDB J.*, 18(2):385–406.

Agrawal, S., Chaudhuri, S., and Das, G. (2002). Dbxplorer: A system for keyword-based search over relational databases. In DBL (2002), pages 5–16.

Alkhateeb, F., Baget, J.-F., and Euzenat, J. (2009). Extending sparql with regular expression patterns (for querying rdf). *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):57–73.

Aluç, G., Hartig, O., Özsu, M. T., and Daudjee, K. (2014). Diversified stress testing of RDF data management systems. In *13th International Semantic Web Conference (ISWC'14)*, volume 8796 of *Lecture Notes in Computer Science*, pages 197–212, Trentino, Italy. Springer.

Aluc, G., Özsu, M. T., and Daudjee, K. (2014). Workload matters: Why RDF databases need a new design. *PVLDB*, 7(10):837–840.

Amer-Yahia, S., Botev, C., and Shanmugasundaram, J. (2004). Texquery: a full-text search extension to xquery. In DBL (2004), pages 583–594.

Angles, R., Boncz, P. A., Larriba-Pey, J.-L., Fundulaki, I., Neumann, T., Erling, O., Neubauer, P., Martínez-Bazan, N., Kotsev, V., and Toma, I. (2014). The linked data benchmark council: a graph and rdf industry benchmarking effort. *SIGMOD Record*, 43(1):27–31.

Angles, R. and Gutiérrez, C. (2008). Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39.

Aranda, C. B., Hogan, A., Umbrich, J., and Vandenbussche, P. (2013). SPARQL web-querying infrastructure: Ready for action? In DBL (2013), pages 277–293.

Arenas, M., Bertails, A., Prud'hommeaux, E., and Sequeda, J. (2012). A direct mapping of relational data to rdf. *W3C Recommendation*.

Arias, M., Fernández, J. D., Martínez-Prieto, M. A., and de la Fuente, P. (2011). An empirical study of real-world sparql queries. *CoRR*, abs/1103.5043. 1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011).

Arthur, D. and Vassilvitskii, S. (2007). k-means++: the advantages of careful seeding. In *18th Annual Symposium on Discrete Algorithms (ACM-SIAM'07)*, pages 1027–1035, New Orleans, Louisiana, USA. SIAM.

Barbieri, D. F., Braga, D., Ceri, S., Valle, E. D., and Grossniklaus, M. (2009). C-sparql: Sparql for continuous querying. In *18th International Conference on World Wide Web (WWW'09)*, pages 1061–1062, Madrid, Spain. ACM.

Becchi, M. and Crowley, P. (2007). An improved algorithm to accelerate regular expression evaluation. In *ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS'07)*, pages 145–154, Orlando, Florida, USA. ACM.

Becchi, M. and Crowley, P. (2008). Extending finite automata to efficiently match perl-compatible regular expressions. In *2008 ACM Conference on Emerging Network Experiment and Technology (CoNEXT'08)*, page 25, Madrid, Spain. ACM.

Berners-Lee, T. (2006). Design issues: Linked data.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):28–37.

Bizer, C., Heath, T., and Berners-Lee, T. (2009a). Linked data - the story so far. *International Journal On Semantic Web and Information Systems*, 5(3):1–22.

Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. (2009b). Dbpedia - a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165.

Bizer, C. and Schultz, A. (2009). The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24.

Boncz, P. A., Erling, O., and Pham, M. (2014). Advances in large-scale RDF data management. In *Linked Open Data - Creating Knowledge Out of Interlinked Data - Results of the LOD2 Project*, volume 8661 of *Lecture Notes in Computer Science*, pages 21–44. Springer.

Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying rdf and rdf schema. In *1st International Semantic Web Conference (ISWC'02)*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68, Sardinia, Italy. Springer.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: Implementing the semantic web recommendations. In DBL (2004), pages 74–83.

Chan, C. Y., Garofalakis, M. N., and Rastogi, R. (2003). Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12(2):102–119.

Chen, T. (2012). *Indexing Text Documents for Fast Evaluation of Regular Expressions*. PhD thesis, Madison, WI, USA. AAI3524345.

Cho, J. and Rajagopalan, S. (2002). A fast regular expression indexing engine. In DBL (2002), pages 419–430.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

Coronel, C., Morris, S., and Rob, P. (2009). *Database Systems: Design, Implementation, and Management*. BPA Series. Cengage Learning, 9th edition.

Cyganiak, R. (2005). A relational algebra for sparql. Technical Report HPL-2005-170.

d'Aquin, M., Baldassarre, C., Gridinoc, L., Angeletou, S., Sabou, M., and Motta (2008). Characterizing knowledge on the semantic web with watson. In *5th International Workshop on Evaluation of Ontologies and Ontology-based Tools (EON'07)*, volume 329, pages 1–10, Busan, Korea. CEUR-WS.org.

Das, S., Sundara, S., and Cyganiak, R. (2012). R2rml: Rdb to rdf mapping language. *W3C Recommendation*.

DeBrabant, J., Pavlo, A., Tu, S., Stonebraker, M., and Zdonik, S. B. (2013). Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953.

Ding, L., Finin, T. W., Joshi, A., Pan, R., Cost, R. S., Peng, Y., Reddivari, P., Doshi, V., and Sachs, J. (2004). Swoogle: a search and metadata engine for the semantic web. In *ACM CIKM International Conference on Information and Knowledge Management (CIKM'04)*, pages 652–659, Washington DC, USA. ACM.

Dorneles, C. F., Gonçalves, R., and dos Santos Mello, R. (2011). Approximate data instance matching: a survey. *Knowledge and Information Systems*, 27(1):1–21.

Duan, S., Kementsietsidis, A., Srinivas, K., and Udrea, O. (2011). Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*, pages 145–156, Athens, Greece. ACM.

Ellul, K., Krawetz, B., Shallit, J., and Wang, M. (2004). Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 9(2/3):233–256.

Elmasri, R. and Navathe, S. B. (2000). *Fundamentals of Database Systems, 3rd Edition*. Addison-Wesley-Longman.

Erling, O. (2006). Advances in virtuoso rdf triple storage (bitmap indexing). Technical report, Openlink Software Inc.

Fan, Y., Lai, W., and Meng, X. (2014). Optimizing database operators by exploiting internal parallelism of solid state drives. *IEEE Data Engineering Bulletin*, 37(2):12–18.

Faro, S. and Lecroq, T. (2013). The exact online string matching problem: A review of the most recent results. *ACM Computing Surveys*, 45(2):13.

Faye, D., CURE, O., and BLIN, G. (2012). A survey of rdf storage approaches. *Revue Africaine De La Recherche en Informatique Et Mathématiques Appliquées*, 15(2012):11–35.

Feigenbaum, L., Williams, G. T., Clark, K. G., and Torres, E. (2013). Sparql 1.1 protocol. *W3C Recommendation*.

Fitzpatrick, B. (2004). Distributed caching with memcached. *Linux Journal*, 2004(124):5–.

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (2012). Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4:1–4:22.

Garbis, G., Kyzirakos, K., and Koubarakis, M. (2013). Geographica: A benchmark for geospatial rdf stores (long version). In DBL (2013), pages 343–359.

Garcia-Molina, H., Ullman, J. D., and Widom, J. (2009). *Database systems - the complete book (2. ed.)*. Pearson Education.

Graefe, G. (2011). Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402.

Gruber, H. and Holzer, M. (2014). From finite automata to regular expressions and back-a summary on descriptional complexity. In *14th International Conference on Automata and Formal Languages (AFL'14)*, volume 151 of *EPTCS*, pages 25–48, Szeged, Hungary.

Guo, Y., Pan, Z., and Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182.

Haase, P., Broekstra, J., Eberhart, A., and Volz, R. (2004). A comparison of rdf query languages. In *3rd International Semantic Web Conference (ISWC'04)*, volume 3298 of *Lecture Notes in Computer Science*, pages 502–517, Hiroshima, Japan. Springer.

Halpin, H., Hayes, P. J., McCusker, J. P., McGuinness, D. L., and Thompson, H. S. (2010). When owl:sameas isn't the same: an analysis of identity in linked data. In *9th International Semantic Web Conference (ISWC'10)*, volume 6496 of *Lecture Notes in Computer Science*, pages 305–320, Shanghai, China. Springer.

Hamilton, J. R. and Nayak, T. K. (2001). Microsoft sql server full-text search. *IEEE Data Engineering Bulletin*, 24(4):7–10.

Harris, S. and Gibbins, N. (2003). 3store: Efficient bulk rdf storage. In *1st International Workshop on Practical and Scalable Semantic Systems (PSSS1)*, volume 89, pages 1–15, Sanibel Island, Florida, USA. CEUR-WS.org.

Harris, S., Lamb, N., and Shadbolt, N. (2009). 4store: The design and implementation of a clustered rdf store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS'09)*, pages 94–109, Washington DC, USA. CEUR-WS.org.

Harth, A., Umbrich, J., Hogan, A., and Decker, S. (2007). Yars2: A federated repository for querying graph structured data from the web. In *6th International Semantic Web Conference (ISWC'07), 2nd Asian Semantic Web Conference (ASWC'07)*, volume 4825 of *Lecture Notes in Computer Science*, pages 211–224, Busan, Korea. Springer.

Hendler, J. A. (2001). Agents and the semantic web. *IEEE Intelligent Systems*, 16(2):30–37.

Hert, M., Reif, G., and Gall, H. C. (2011). A comparison of rdb-to-rdf mapping languages. In *7th International Conference on Semantic Systems, (I-SEMANTICS'11)*, ACM International Conference Proceeding Series, pages 25–32, Graz, Austria. ACM.

Hogan, A., Harth, A., Umbrich, J., Kinsella, S., Polleres, A., and Decker, S. (2011). Searching and browsing linked data with swse: The semantic web search engine. *Journal of Web Semantics*, 9(4):365–401.

Huang, J., Abadi, D. J., and Ren, K. (2011). Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134.

Jindal, V., Bawa, S., and Batra, S. (2014). A review of ranking approaches for semantic search on web. *Information Processing and Management*, 50(2):416–425.

Kämpgen, B. and Harth, A. (2013). No size fits all - running the star schema benchmark with sparql and rdf aggregate views. In *10th International Conference (ESWC'13)*, volume 7882 of *Lecture Notes in Computer Science*, pages 290–304, Montpellier, France. Springer.

Kleene, S. C. (1951). Representation of events in nerve nets and finite automata. *Automata Studies*.

Klyne, G., Carroll, J. J., and McBride, B. (2004). Resource description framework (rdf): Concepts and abstract syntax. *W3C Recommendation*.

Knuth, D. E., Jr., J. H. M., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350.

Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., and Turner, J. S. (2006). Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'06)*, pages 339–350, Pisa, Italy. ACM.

Lee, J., Kasperovics, R., Han, W., Lee, J., Kim, M., and Cho, H. (2015). An efficient algorithm for updating regular expression indexes in RDF databases. *International Journal of Data Mining and Bioinformatics (IJDMB)*, 11(2):205–222.

Lee, J., Pham, M., Lee, J., Han, W., Cho, H., Yu, H., and Lee, J. (2011). Processing SPARQL queries with regular expressions in RDF databases. *BMC Bioinformatics*, 12(S-2):S6.

Lee, S., Moon, B., Park, C., Kim, J., and Kim, S. (2008). A case for flash memory ssd in enterprise database applications. In *2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, pages 1075–1086, Vancouver, Canada. ACM.

Li, Y., He, B., Yang, J., Luo, Q., and Yi, K. (2010). Tree indexing on solid state drives. *PVLDB*, 3(1):1195–1206.

Losemann, K. and Martens, W. (2013). The complexity of regular expressions and property paths in sparql. *ACM Transactions on Database Systems (TODS)*, 38(4):24.

Minack, E., Sauermann, L., Grimnes, G., Fluit, C., and Broekstra, J. (2008). The sesame lucenesail: Rdf queries with full-text search. Technical report.

Minack, E., Siberski, W., and Nejdl, W. (2009). Benchmarking fulltext search performance of RDF stores. In *6th European Semantic Web Conference, (ESWC'09)*, volume 5554 of *Lecture Notes in Computer Science*, pages 81–95, Heraklion, Crete, Greece. Springer.

Morsey, M., Lehmann, J., Auer, S., and Ngomo, A.-C. N. (2011). Dbpedia sparql benchmark - performance assessment with real queries on real data. In DBL (2011), pages 454–469.

Morsey, M., Lehmann, J., Auer, S., and Ngomo, A.-C. N. (2012). Usage-centric benchmarking of rdf triple stores. In *AAAI*, Toronto, Ontario, Canada. AAAI Press.

Namjoshi, K. S. and Narlikar, G. J. (2010). Robust and fast pattern matching for intrusion detection. In *29th IEEE International Conference on Computer Communications (INFOCOM'10)*, pages 740–748, San Diego, CA, USA. IEEE.

Neumann, T. and Moerkotte, G. (2011). Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *27th International Conference on Data Engineering (ICDE'11)*, pages 984–994, Hannover, Germany. IEEE Computer Society.

Neumann, T. and Weikum, G. (2008). RDF-3X: a risc-style engine for RDF. *PVLDB*, 1(1):647–659.

Ngomo, A. N. and Schumacher, F. (2009). Borderflow: A local graph clustering algorithm for natural language processing. In *10th International Conference on Computational Linguistics and Intelligent Text Processing (CICLing'09)*, volume 5449 of *Lecture Notes in Computer Science*, pages 547–558, Mexico City, Mexico. Springer.

Ngonga Ngomo, A.-C. and Röder, M. (2016). HOBBIT: Holistic benchmarking for big linked data. In *ESWC, EU networking session*.

Nitta, K. and Savnik, I. (2014). Survey of rdf storage managers. In *6th International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA'14)*, pages 148–153, Chamonix, France. IARIA.

Paliwal, K. K., Sharma, A., Lyons, J., and Dehzangi, A. (2014). A tri-gram based feature extraction technique using linear probabilities of position specific scoring matrix for protein fold recognition. *IEEE Transactions on NanoBioscience*, 13(1):44–50.

Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3).

Prud'Hommeaux, E. and Seaborne, A. (2008). Sparql query language for rdf. *W3C Recommendation*.

Ratnasamy, S., Stoica, I., and Shenker, S. (2002). Routing algorithms for dhts: Some open questions. In *International workshop on Peer-To-Peer Systems (IPTPS'02)*, volume 2429 of *Lecture Notes in Computer Science*, pages 45–52, Cambridge, MA, USA. Springer.

Reidenbach, D. and Schmid, M. L. (2011). A polynomial time match test for large classes of extended regular expressions. In *15th International Conference on Implementation*

and Application of Automata (CIAA'10), volume 6482 of *Lecture Notes in Computer Science*, pages 241–250, Winnipeg, MB, Canada. Springer.

Sahoo, S., Halb, W., Hellmann, S., Idehen, K., Thibodeau, Jr, T., Auer, S., Sequeda, J., and Ezzat, A. (2009). A survey of current approaches for mapping of relational databases to rdf. *W3C RDB2RDF Incubator Group Report*.

Saleem, M., Mehmood, Q., and Ngomo, A. N. (2015). FEASIBLE: A feature-based SPARQL benchmark generation framework. In *14th International Semantic Web Conference (ISWC'15)*, volume 9366 of *Lecture Notes in Computer Science*, pages 52–69, Bethlehem, Pennsylvania, USA. Springer.

Samet, H. (1990). Data structures for databases. In *Applications of Spatial Data Structures*, page 507. Addison-Wesley / Helix Books.

Saxena, R., Singh, K., and Jaiswal, U. (2011). A fast sentence searching algorithm. In *2nd International Conference on Advances in Communication, Network, and Computing (CNC'11)*, pages 557–561, Bangalore, India. Springer Berlin Heidelberg.

Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., and Tran, T. (2011). Fedbench: A benchmark suite for federated semantic data query processing. In DBL (2011), pages 585–600.

Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. (2008). Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627.

Schmidt, M., Meier, M., and Lausen, G. (2010). Foundations of SPARQL query optimization. In *13th International Conference on Database Theory (ICDT'10)*, pages 4–33, Lausanne, Switzerland. ACM.

Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011). Fedx: Optimization techniques for federated query processing on linked data. In DBL (2011), pages 601–616.

Seaborne, A. (2014). Private Communication [email].

Seaborne, A., Prud'Hommeaux, E., and Harris, S. (2013). Sparql 1.1 query language. *W3C Recommendation*.

Shadbolt, N., Hall, W., hendler, J., and Dutton, W. H. (2013). Web science: a new frontier. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1987).

Silberschatz, A., Korth, H. F., and Sudarshan, S. (2010). *Database System Concepts*. McGraw-Hill Education, 6th edition.

Smith, R., Estan, C., and Jha, S. (2008). Xfa: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy (S&P'08)*, pages 187–201, Oakland, California, USA. IEEE Computer Society.

Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., and Reynolds, D. (2008). SPARQL basic graph pattern optimization using selectivity estimation. In *17th International Conference on World Wide Web (WWW'08)*, pages 595–604, Beijing, China. ACM.

Stonebraker, M. and Çetintemel, U. (2005). "one size fits all": An idea whose time has come and gone. In *21st International Conference on Data Engineering (ICDE'05)*, pages 2–11, Tokyo, Japan. IEEE Computer Society.

Stuckenschmidt, H., Vdovjak, R., Houben, G.-J., and Broekstra, J. (2004). Index structures and algorithms for querying distributed rdf repositories. In DBL (2004), pages 631–639.

Stuhr, M. and Veres, C. (2013). Filt - filtering indexed lucene triples - a sparql filter query processing engine. In *10th International Conference on Flexible Query Answering Systems (FQAS'13)*, volume 8132 of *Lecture Notes in Computer Science*, pages 25–39, Granada, Spain. Springer.

systap (2013). Bigdata by systap. Technical report, SYSTAP, LLC.

Thompson, K. (1968). Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.

Tripathy, A., Agrawal, A., and Rath, S. K. (2016). Classification of sentiment reviews using n-gram machine learning approach. *Expert Systems with Applications*, 57:117–126.

Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V., and Boncz, P. A. (2012). Heuristics-based query optimisation for SPARQL. In *15th International Conference on Extending Database Technology (EDBT'12)*, pages 324–335, Berlin, Germany. ACM.

Tsirogiannis, D., Harizopoulos, S., Shah, M. A., Wiener, J. L., and Graefe, G. (2009). Query processing techniques for solid state drives. In Çetintemel, U., Zdonik, S. B., Kossmann, D., and Tatbul, N., editors, *2009 ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*, pages 59–72, Providence, Rhode Island, USA. ACM.

Tummarello, G., Delbru, R., and Oren, E. (2007). Sindice.com: Weaving the open linked data. In *6th International Semantic Web Conference (ISWC'07), 2nd Asian Semantic Web Conference (ASWC'07)*, volume 4825 of *Lecture Notes in Computer Science*, pages 552–565, Busan, Korea. Springer.

Wang, X., Wang, S., Du, P., and Feng, Z. (2010). Storing and indexing rdf data in a column-oriented dbms. In *2nd International Workshop on Database Technology and Applications (DBTA'10)*, pages 1–4, Wuhan, China. IEEE.

Wei, W., Barnaghi, P. M., and Bargiela, A. (2008). Search with meanings: an overview of semantic search systems. *International Journal Communications of SIWN*, 3:76–82.

Wilkinson, K., Sayers, C., Kuno, H., and Reynolds, D. (2003). Efficient rdf storage and retrieval in jena2. In *1st International Workshop on Semantic Web and Databases (SWDB'03)*, pages 131–150, Berlin, Germany. Online Publication.

Yang, L., Karim, R., Ganapathy, V., and Smith, R. (2011). Fast, memory-efficient regular expression matching with nfa-obdds. *Computer Networks*, 55(15):3376–3393.

Yu, F., Chen, Z., Diao, Y., Lakshman, T. V., and Katz, R. H. (2006). Fast and memory-efficient regular expression matching for deep packet inspection. In *ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS'06)*, pages 93–102, San Jose, California, USA. ACM.

Zhang, Y., Pham, M.-D., Corcho, Ó., and Calbimonte, J.-P. (2012). Srbench: A streaming rdf/sparql benchmark. In *11th International Semantic Web Conference (ISWC'12)*, volume 7649 of *Lecture Notes in Computer Science*, pages 641–657, Boston, MA, USA. Springer.