University of Zurich[UZH]

SPARQL Query Approximation With Bloom Filters

Bachelor Thesis June 23, 2016

Stefanie Ziltener
of Buttikon SZ, Switzerland

Student-ID: 11-727-856
stefanie.ziltener@uzh.ch

Advisor: **Tobias Grubenmann**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to express my sincere gratitude to my supervisor Tobias Grubenmann for his support, patience and guidance during my thesis. I would like to thank Prof. Abraham Bernstein for giving me the opportunity to write this thesis at the Department of Informatics at the University of Zurich.
Finally, I want to thank everyone whose support I received in one or the other way during the time of this thesis; especially my partner and family.

Zusammenfassung

Das Thema dieser Arbeit ist die Abschätzung der Resultate von SPARQL-Query Abfragen auf RDF Daten. In Standard-Datenbankumgebungen wird 'Approximate Query Processing' oft benutzt, um die Resultate zu schätzen. Ein Beispiel, wo sich die Alternative zur exakten Ausführung des Queries lohnen kann ist, wenn Ressourcen eingeschränkt sind - wie die Rechenkapazität, Speicherplatz, Zugriff zur Datenbank oder Geld. Die Annäherung des Resultats kann als Entscheidungsgrundlage für weitere Überlegungen zu einer solchen Informationsabfrage-Strategie dienen.
In der Arbeit wird eine Methode aus drei vorgestellten Vorgehen ausgewählt, um die Abschätzung in den gewünschten Semantic Web Kontext zu transferieren. Der gewählte Algorithmus benutzt Bloom Filter, um Zwischenresultate darzustellen und sie dann für das Schlussresultat miteinander zu vereinen. Die Implementierung des Algorithmus wurde in Java geschrieben und dann in Experimenten mit Betrachtung der Laufzeit und des relativen Abschätzungsfehlers ausgewertet. Die Analyse der Daten zeigt, dass die Methode noch nicht genügend optimiert ist, um generell positive Resultate zu liefern. Schlussendlich wird ein Fazit gezogen, das Limitierungen und Verbesserungsideen des Abschätzungsprozesses präsentiert und einen Ausblick auf zukünftige Arbeit zeigt.

Abstract

The topic of this thesis is SPARQL query approximation on RDF data. In standard database contexts, using approaches for approximating query results is common. An example of a motivation for using query approximation instead of accurate execution is that resources in the form of computing power, disk space, money, and database access can be restricted. Approximating the query results can serve as a decision basis for or against further processing of a querying strategy.
The thesis analyses an approach to transfer one of three presented methods for query approximation to the Semantic Web context. The chosen algorithm uses Bloom filters to represent datasets of query conditions and additionally to join the sub results for the result approximation. The algorithm was implemented in Java code and compared to the actual query execution on the aspects of runtime and relative error of the results. The evaluation has shown that the approach is not yet sufficiently elaborated for overall positive results. With the limitations and optimization ideas that are presented, a conclusion is drawn with an outlook to future work.

Table of Contents

1

Introduction

Data is everywhere - and the capability to store it on hardware has grown heavily in the last years. It was estimated that in the year 2007, 290 optimally compressed Exabytes could be stored with analogue and digital technology worldwide, with the digital part being dominant [Hilbert and López, 2011]. It is clear that in the meantime, that size has grown - how much can only be speculated. With this development, the term *Big Data* has emerged; it is used *"to refer to the increase in the volume of data that are difficult to store, process, and analyse through traditional database technologies"* [Hashem et al., 2015].
Dealing with this lot of data, i.e. to make it searchable and interpretable, is an important task. With normal query execution, as there is more data stored in a database, generally the queries take longer to execute. This can be countered partially with a good and up-to-date index on the data. To execute a query on a huge data set without an index is time-consuming (as can be seen in a specific context e.g. in [Jamard and Gardarin, 2007]). Often times, data from distributed databases has to be joined to get a complete result. In the case of *Linked Data*, it can happen that these data sets are huge; and so the network load is high for sending them all to a central place to be joined.

Approximate Query Processing (AQP) is used to estimate the result of a database query. It generally trades off the correctness of the result for faster execution time. This can be a good alternative to exact query execution when either the database to be queried is huge, the runtime of the query is crucial, or when both constraints come together. It can be used as an indication for the decision whether the actual query is worth executing or not. In the environment of traditional databases, estimating the result of queries is a procedure that is well-known and implemented in many use cases (for example, Oracle uses it to optimize query plans [Dell'Era, 2007]). Some approaches for AQP that have been taken and will be presented in this thesis include the following: Sampling the data set and conducting the query on the sample [Agarwal et al., 2014], creating histograms of the data [Muralikrishna and DeWitt, 1988] or using *Bloom filters*: [Bloom, 1970] and [Papapetrou et al., 2010].
One use case where AQP can be meaningful is with data from the *Semantic Web* (see Section 2.1). This data has overall a large size and is distributed, which can cause many joins of data sets for queries. More detailed insight is provided in the paper "What is the Size of the Semantic Web", which includes a section that shows the size of several

data dumps of different endpoints [Hausenblas et al., 2008].
In this thesis, an approach is taken to estimate queries on data of the Semantic Web and compare the results to normal query execution. The chosen algorithm will be implemented in a Java program and experimentally tested for result correctness and runtime values.

1.1 Motivation

The aim of this thesis is to find and test out a method for query approximation in a specific context. For that, firstly, three selected approaches to approximate queries will be presented and analysed for suitability. From these approaches, one is chosen to be implemented in Java source code. This is done to evaluate in experiments how well our derived algorithm compares to the actual query execution.
The setting for our approach is a query execution across distributed databases, each with thematically consistent data in the form of *RDF* triples. The objective of the query approximation is to find out how many results a specific *SPARQL* query has. SPARQL is a querying language similar to SQL, but especially built for Semantic Web data [W3C Recommendation, 2008] and is introduced in Section 2.1. Our approach uses Bloom filters to approximate selective data JOINS. Several other approaches were considered for usage but withdrawn because of practical reasons that will be explained in Section 2.2. Bloom filters can store a large amount of data space efficiently, and then perform membership tests on further data. They are efficient for cases where only a small quantity of elements qualifies for membership in the set that formed the Bloom filter. Thus, it is intuitively a suitable approach for queries in the Semantic Web, as often the result size of a query is immensely smaller than the size of the queried dataset.
With an estimate of the result size of the query, it can be decided if the exact query execution is worth the actual execution. This could be interesting for cases where we don't actually know if the query is formulated too loose and has too many results. Another example is, when several queries are generated to find out a specific information and the user only wants to execute the query with the most results. It could further be used as a decision-making help when, for example, information is not freely accessible and we have to pay with some resources to get an accurate answer. For deciding if a query should be executed, a close enough estimation of the result cardinality serves the purpose in these cases. Resources like time (assuming the approximation is faster than the exact execution), money (if database access is costly) or computing power can be saved if the query does not have to actually be executed.
The experiment compares the approximation approach with Bloom filters against a complete calculation of results. For this, the algorithm is run with different input parameters that decide on the Bloom filter accuracy. The test set consists of 20 different queries. The result will be analysed and compared with respect to the correctness of the result and the computing time used for each approach.

1.2 Thesis Outline

In the following Chapter 2 an outlook to some related work will be presented.
Chapter 3 introduces the definition and explanation of some basic information that is important for the understanding of this thesis' contents. This covers specifically the basics of the Semantic Web, Linked Data, RDF, and SPARQL. In Section 3.2 the Bloom filter is introduced.
Then, in Chapter 4, different methods of approximating queries are shown. Section 4.1 discusses the pro and cons they would have in this context. Furthermore, the reason for choosing the Bloom filter for our approach is explained. The algorithm used for the query approximation is presented in Section 4.2. Finally, the expectations for the performance of the implemented algorithm in the experiments are shown in Section 4.3.
The implementation of the algorithm is done in Chapter 5. In Section 5.1, the basic Java code structure is explained. The actual purpose and objective of the experiments with respect to the expectations is highlighted in Section 5.2. At last, the setup of the experiments will be presented in Section 5.3.
The following Chapter 6 covers the evaluation of the results. In Section 6.1, the results are presented and discussed. Additionally, some limitations concerning this thesis and points for future work will be discussed in Section 6.2.
Finally, conclusions are drawn in Chapter 7.

2

Related Work

There is some work already done in the area of RDF Query Processing. In this section, I will present an outlook on such related work.
In the first paper **An Evolutionary Perspective on Approximate RDF Query Answering** a framework is created that uses an evolutionary algorithm [Guéret et al., 2008]. The objective of the approach is approximating queries with imperfect information; be it uncertain data or users not knowing how to formulate the queries for the information they look for, which could result in inadequate queries. The authors *"propose a different approach which consists of, iteratively, guessing a set of complete assignments for the query variables (a "candidate solution"), verifying those assignments, and if no solutions are found, loop and trying again"* [Guéret et al., 2008]. With further iterations, the algorithm should perform better and converge towards an approximate solution. For verifying in a fast way if the candidates fit the constraints, Bloom filters are used. The method can deal with uncertainty of data, approximate answers and approximate queries. Furthermore, in the conclusions, it has a positive outlook to scalability.
A different approach is taken in the paper **Efficient approximate SPARQL querying of Web of Linked Data** [Reddy and Kumar, 2010]. The algorithm uses ontologies published in the web to approximate SPARQL queries. Based on information of the ontology, the query gets relaxed at runtime. With this approach, they extend previous work already done that relaxes queries [Huang et al., 2008].
Another paper that relies on ontologies to approximate queries is **Searching the Semantic Web: Approximate Query Processing based on Ontologies** by [Corby et al., 2006]. This method is based on the assumption that users that want to query the data can have differing viewpoints to those of designers of ontologies and query languages. This misunderstanding could lead to inefficient querying of the Semantic Web. The paper presents its own search engine for querying the Semantic Web, and introduces with it a new query language.
These approaches all differ in their method of querying Linked Data. However, they are all trying to optimize query execution by approximating the resulting value. To the knowledge of the author of this thesis, there has been made no comparison of using plain Bloom filters to approximate SPARQL queries to the correct query computation. The contribution of this thesis includes the creation of an approximation algorithm, the implementation of it in Java and the comparison to the exact query execution.

3

Background

In this chapter, I will give a short overview on background information to some of the topics that are related to this thesis. It will only cover the basics needed to gain an introductory insight into the field of the Semantic Web, Linked Data, RDF and SPARQL. This is done to help to understand the following chapters of the thesis.

3.1 Semantic Web and RDF

According to Tim Berners-Lee, one of the people who greatly influenced the term, the Semantic Web is an extension of the known Web constructed to make sharing and searching for data easier. The Web in its current state is optimized to human consumption. In contrast, searching and sharing in the Semantic Web should be equally optimal for humans and computers. One of the key concepts was meant to bring structure to the web, to link data in a well-defined manner producing a graph spanning over different data sources. Ideally, this would produce a connection over all data in the Semantic Web. This serves the goal that computers could search for, process, and interpret semantic content without having to possess some high scaled artificial intelligence or having to implement complicated algorithms. The structure would provide the context for information that we, as humans, can easily understand. An example for this would be if one decided to search for all male names that start with a B. When it is not defined, a computer does not know if a word is a persons' name instead of a towns', for example. In this context, information (like a name), would have a kind of tag that further defines "what" it is - and thus it would be easy to filter for names of a person. Since tags can be invented by anyone, they have to be given a more comprehensive meaning that sets it in a global context. These objectives are served with different approaches, for example with *XML* (to create tags) and RDF (to give the tags a general context). In this example, it would be simple to add the requirement that it had to be a male name. Since all data is connected in this ideal, it would be just another join clause in a query. [Berners-Lee et al., 2001]
Linked Data is a term that evolved from the Semantic Web ideology, it was also coined by Tim Berners-Lee. It is one of the core concepts of the Semantic Web ideal. It uses RDF to connect information from different sources. [Berners-Lee, 2006]

According to the *World Wide Web Consortium (W3C)*, the Resource Description Framework or RDF *"is a directed, labelled graph data format for representing information in the Web"* [W3C Recommendation, 2008]. In an RDF graph, there are three different node types which are connected by predicates, serving as edges in the graph. For simplicity, we are omitting the, for this thesis, irrelevant parts of the RDF syntax. Interesting for us are the literal nodes and the reference nodes. Both represent resources, in case of the literal it can be any text string, for the reference node it has to be a *URI*. An RDF triple consists of two nodes and a predicate that represents the uni-directional relation between the two nodes. It can be seen as a statement in the form of "Subject Predicate Object", e.g. the triple

```
<http://data.linkedmdb.org/resource/film/2820>
<http://purl.org/dc/terms/title>
Tarzan
```

can be read as: "The film with id 2820 has a title which is Tarzan". An excerpt of what a part of the graph could look like is seen in Figure 3.1.

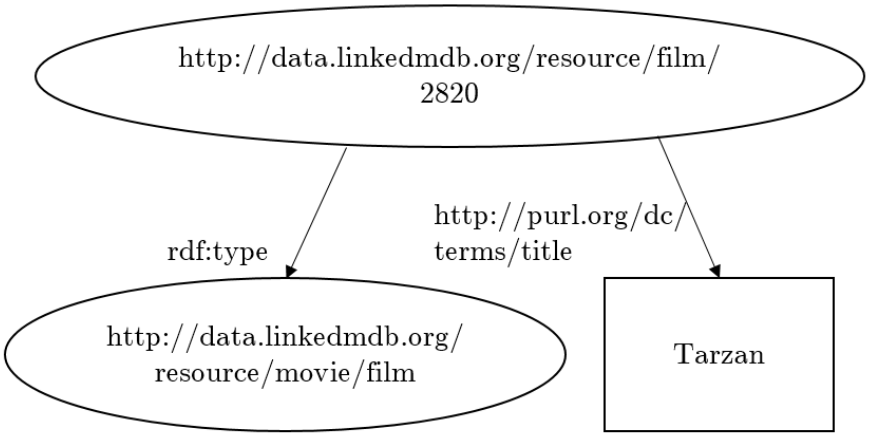

Figure 3.1: Example part of an RDF graph

Since as much data as possible is linked in the Semantic Web (thus the term Linked Data), the graph is in reality much more extended. The predicate is denoted as the edge that connects two nodes. Here, we have two different node types that differ in the way they are shown in the graph. "Tarzan" is a literal and as such inside a square. The other two nodes are URIs and with this, reference nodes and drawn inside a circle.
The RDF query language that is used in this thesis is SPARQL. In the thesis, only SPARQL Select Queries have been used for the evaluation. A typical SPARQL Select Query can be formulated similar to:

```
SELECT ?title ?actor WHERE {
  ?film <http://data.linkedmdb.org/resource/movie/actor> ?actor .
  ?film <http://purl.org/dc/terms/title> 'Tarzan'
};
```

Listing 3.1: Example SPARQL Query

The dot '.' serves the same purpose as an AND term in a SQL query. SPARQL generally was designed to have a similar syntax to SQL, so a further explanation of this is omitted. For further information on SPARQL, the W3C has made a reference website [W3C Recommendation, 2008].

3.2 Bloom filter

A Bloom filter is a structure to store data in a space-efficient manner and test whether elements are included in the stored data set or not [Bloom, 1970]. It is a term coined after Burton H. Bloom. He presented the concept of the filter in his paper "Space / Time trade-offs in hash coding with allowable errors" and the following section elaborates further on this description of the Bloom filter. The structure consists of an array of ***m bits***, initially all set to 0 as a symbol for an empty data set. To build the filter, each element from the data to be stored gets hashed. For this, ***k different hash functions*** are applied. The ***k results*** of the hash functions each correspond to an index in the array. The content of these resulting array indices is then set to 1. After all elements are processed this way, the resulting structure represents the data set and additionally serves as a filter for performing membership tests on the set. To perform a membership test on an element, it gets hashed just like the stored data before. The bits in the filter that correspond to the hash result are then inspected - if all have been previously set to 1, the element could be a member in the set. If even one bit in the filter is 0, the tested element is definitely not contained in the set.
The construction of the Bloom filter allows no false negatives, but there is a possibility for false positives. A false negative means that a value that is actually a member of the set would be rejected by the membership test with the Bloom filter. This means, in the case of the Bloom filter we can be sure that when the test has a negative result, the value is definitely not included in the set. A false positive means the opposite; that an element that actually does not exist in the set gets a positive result with a membership test. The reason for this being possible in the Bloom filter is because of the ***k indices*** getting put to 1 for a single element insertion. With the filter getting fuller, it is more possible that a certain combination of indices set to 1 overlap in such a way that it fits the ***k hash function results*** of a false positive element. This possibility (called *false positive probability (FPP)*) depends on the size of the filter, the fullness of the filter and the number of used hash functions. This dependency is shown with the following formulas from the paper "Compressed Bloom Filters" [Mitzenmacher, 2001]. According to this, the following formulas apply: After we have put all ***n elements*** into the filter with ***k hash functions*** and total ***size of the filter m***, the probability that a bit in

the filter is set to 1 is:

$$p = 1 - \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}} \tag{3.1}$$

From equation 3.1 Bloom deducted that the FPP is equal to:

$$p^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(e^{-\frac{kn}{m}}\right)^k \tag{3.2}$$

As the size of the array is increased, the FPP can be decreased. At the same time though the space efficiency of the Bloom filter is decreased, so it is a trade-off that must be considered for the experimentation. [Bloom, 1970]
In practice, the Bloom filter is applied in several different use cases. It is primarily efficient for cases where only a small part of a large chunk of data is in the defined set when testing for membership. One example for practical usage of Bloom filters is identifying malicious *URLs* when they are entered in the web browser. The Google Chrome browser used to keep a local Bloom filter for checking URLs against, and when a URL matched the filter and thus was possibly malicious, a full check of it was performed [Yakunin, 2010]. Bloom filters can also be used in databases. For example, Google BigTable and Apache HBase use Bloom filters to check if a row or column exists in a data set and thus reduces unnecessary disk access for those that do not match the filter [Chang et al., 2006]. Another environment where Bloom filters are used is *Peer to Peer* systems. Bitcoin uses Bloom filters to filter and verify payment transactions faster [Antonopoulos, 2014]. B-Trackers use Bloom filters *"to avoid peers discovering providers they already know"* [Hecht, 2011].

4

Query Approximation Method

This chapter presents a small selection of query approximation methods. These methods were mostly meant to be used in a traditional database environment, so they likely have to be adapted to fit the RDF data concept. It will be analysed how well the approach fit the desired context. Additionally, it will be chosen what method is going to be implemented and evaluated in this thesis.

4.1 Candidates

In this chapter three methods of query approximation in databases will be presented. They will then be analyzed for their suitability for the Semantic Web data we want to use it on. After that, the chosen method is explained in more detail. At last, the proposed algorithm will be explained. The first method that is reviewed is the approach using histograms to estimate queries. The essential paper that was reviewed regarding this topic is [Muralikrishna and DeWitt, 1988].
In that paper, **Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries**, we are presented an algorithm with which we can create equi-depth histograms for multi-dimensional data. It then proposes a new storage structure for optimal storage and search of the histogram, the H-tree. Finally, two schemes are presented to show how to estimate queries with the help of these structures and algorithms. These schemes were experimentally tested in two runs. The objective was to find out the maximum error the two schemes had and then, with the second run, to explore the average behavior of the second scheme.
The paper is very thorough in its exploration of a possible query estimation approach. It starts with explaining why equi-depth histograms are used. They produce a better error quote than equi-width histograms with data that is not uniformly distributed. Equi-depth histograms use buckets that all have the same number of elements in them, contrary to equi-width histograms buckets that all cover the same width over the possible range of attributes. With the control of the depth of buckets we can control the maximum estimation error. Then, we are presented an algorithm to generate such equi-depth histograms for multiple attributes. This is then called a multi-dimensional histogram. At the beginning of the algorithm, a decision has to be made on the ***number of total buckets m*** and how to divide that into ***numbers of buckets of the specific i***, which

gives us n_i. The algorithm is structured in such a way that it begins with the bucket that contains all tuples. The tuples are sorted in their pre-constructed bucket along the ***dimension i*** and then partitioned into n_i ***segments***, which form the smaller buckets. The tree structure that is created is a variation derived from the idea of the R-tree. It uses the before mentioned hierarchical partitioning for storage. That means, on each ***level i*** of the tree, we represent the partitioning of the ***i-th dimension***. The final buckets are represented as the leave nodes from the tree. This approach is also used to search for fitting tuples. For each query, a query box is defined. We have to search for overlapping or fitting values in each level of the tree, corresponding to the dimension in the query. It is differentiated between f-buckets, whose content lies completely in the query box, and p-buckets that only overlap the query box. With this search approach, both kind of buckets will be retrieved and they can easily be distinguished from each other. In the two different schemes, the amount of tuples are estimated based on different formulas. The Half Scheme and the Uniform Scheme were then experimentally tested. Additionally to those experiments, the paper presents experiments with the same approach applied on a random sample of the original tuple space. [Muralikrishna and DeWitt, 1988]
An advantage of that approach are that the maximum error of the estimation can be controlled via the sample size and the number of buckets. Furthermore, the approach does not only work with uniformly distributed data, which we do not have in our scope of work. And lastly, it has the advantage that the histogram has to be created once. After that, a simple search in the tree can be used for all queries, and then the estimation is computed with one of the two formulas. This speeds up the query estimation retrieval a lot compared to an approach that has to compute the estimation for each query anew.
A disadvantage is that the approach was tested with multi-dimensional tuples consisting of integer values. Deciding what the dimensions are and how to sort the data is intuitive with integers, but not that much with SPARQL queries and RDF tuples that do have URIs or literals as values. This would require a solution to make the transfer to the problem scope. Moreover, the translation of a SPARQL query to the query box that is used for the approximation is not as simple as it was in the experiments. A further disadvantage that comes to mind is that the histogram is static and can become dated fast. So each time the data is modified, or after a specific amount of time, a new histogram would have to be calculated to guarantee the timeliness of the result. For the scope of the experiments, this would not pose a problem; however, in reality data on the web is modified often.

The second approach is Sampling-based Approximate Query Processing (S-AQP). The more specific explanation of such an approach is explained in **Knowing When You're Wrong: Building Fast and Reliable Approximate Query Processing Systems** [Agarwal et al., 2014]. The focus of this paper lies on estimating the error that we get with approximating queries with different sampling methods. It presents an explanation of basic sampling methods and additionally creates a pipeline architecture that uses their findings in a practical S-AQP example. First, an overview on AQP is given. Sampling is mentioned as *"one of the most common and generic approaches to approxi-*

mation of analytical queries" [Agarwal et al., 2014]. Instead of querying the whole data set, we execute the query over a sample set. The procedure of sampling can be done in different ways; for example, sampling the rows of the database n times at random. Since this sampling is random, it can be said to be drawn from a sampling distribution. This distribution is helpful in providing a confidence interval for the estimated query result. The paper presents three different methods for estimating the sampling distribution and then analyzes the usefulness of these techniques with real data and real queries. The result is that no method is superior in general, and that they have to differentiate the cases to decide which method should be used. This is done with a diagnostic. At this stage, the proposed architecture for approximating queries is presented. It uses a Logical Query Plan that consists of three parts: The query estimation on the sample dataset, the error computation and the part that executes the diagnostic test. These parts can be run in parallel and communicate with each other. They do not run on a single sample, but on several distributed sample sets. In their approach, they added support for a re-sampling operator for when the selected random sample performed badly. This pipelined process was a naïve solution with overhead that unnecessarily slowed down the query approximation. So, a further round of optimization was done to speed up the runtime. The logical query plan, the physical query plan and the storage layer were all adapted to maximize the efficiency of this approximation approach. [Agarwal et al., 2014]
In the end, this approach had great results and sped up the approximate query execution by a factor of 10-200. However, most of the speed up was achieved with specific and quite complex adaptions of not only the algorithm but also the underlying querying framework. For the scope of this thesis, these adaptations would be far too complex and time-consuming to implement. Far more appropriate would be to implement only the basic approach. It consists of sampling the data, querying the sample, controlling the error with the confidence interval and resampling if necessary. Such an approach would be a possible method for query approximation.
An advantage would again be that we can run all the queries on the same sets of sample data. If a group of queries have an unsatisfactory confidence interval, they can be executed over the same resampled sample data. With this, the overhead of the sampling can be softened. However, there is no general sample command in SPARQL comparable to the one used in traditional databases. Although there exists a sample function in SPARQL, the implementation is not obliged to return a random sample. The only requirement in the SPARQL specification is that the sample command has to return an arbitrary value of the set given as input [W3C Recommendation, 2013]. It could be that this is always the first possible value in the set. Moreover, the whole resampling could not be taken over from the paper as this is again standardized for SQL. It would have to be adjusted to the SPARQL query language as good as possible, as SQL has a far more extensive functionality. The rest would have to be newly implemented.
Another advantage this approach has over the histogram approach is that it does not matter which format the data is in. There would have to be no adaptions done because of the fact that the values are literals.

The last presented approach is the query approximation with the help of Bloom filters. The basic information about Bloom filters were given in the identically named Section 3.2. This approach is based on the idea that the query is split into subqueries. These subqueries are then executed separately. Each subquery result set is then transformed into a Bloom filter. This transformation does include the use of other filters calculated in previous steps to simulate a Join of different subqueries. The exact algorithm is explained in detail in the following Section 4.2.
The advantage of using Bloom filters is that the method is quite intuitively rendered to our task domain. It does not have any constraints that speak against the RDF data, e.g. the data does not have to be transformed into a form that would make sorting possible. It also does not implicitly demand any functionality of the query language. We can just enter a plain String as an input value for the Bloom filter. Moreover, the Bloom filter structure seemed to present an easy way of controlling the accuracy of the result. This could be done via the FPP value, which can be adapted by choosing appropriate values for the number of hash functions and the size of the filter (with a given number of element insertions).
A further advantage is that the Bloom filter itself serves as a data structure as well as an integral part of the algorithm, as can be seen in Section 4.2. It makes the algorithm a piece less complex. Also it doubles as a compressed data storage, which brings up the idea of sending it over the network instead of the data chunk. This could be an interesting factor for possible experimentation. The final reason for choosing the Bloom filter method for this thesis is the complexity of the algorithm approach. Also, this method does not restrict its basic usage to a specific data type or query language. That means it did not have to be transferred, which would have again cost time. Since the time for this thesis is restricted, the approach with the least workload has been chosen. With that, the possibility of unpredictable implementation issues during the practical part could be minimized. Bloom filters are well studied and there are complete classes coded for their usage.

4.2 Proposed Algorithm

In this section, an algorithm using Bloom filters to approximate query results is presented. For easier understanding, an example taken from the sample queries that were evaluated and a pseudo code extract is shown. Suppose we have a SPARQL query like Listing 4.1, which searches for all films and their genre that have an Italian director.

```
SELECT ?film ?director ?genre WHERE {
  ?director <http://dbpedia.org/ontology/nationality>
  <http://dbpedia.org/resource/Italy>.
  ?film <http://dbpedia.org/ontology/director> ?director.
  ?x <http://www.w3.org/2002/07/owl#sameAs> ?film.
  ?x <http://data.linkedmdb.org/resource/movie/genre> ?genre.
};
```

Listing 4.1: Example SPARQL Query

In the following context, an unbound variable is seen as a variable that has no pre-set value to it. That means it is a placeholder for various possible values. When the query gets executed, these variables get bound to concrete values that fit the conditions in the query. In SPARQL, the variables are indicated with a question mark ("`?`") in front of their name. An example for a binding to the variable `?director` (with the first subquery as the only context) is `<http://dbpedia.org/resource/Gabriele_Muccino>`. In the first step of the algorithm, each unbound variable gets assigned its own Bloom filter, which is still empty. The query is split up into triple patterns by the dot ("."). So, the first triple pattern looks like this:

```
?director
<http://dbpedia.org/ontology/nationality>
<http://dbpedia.org/resource/Italy>
```

These triple patterns are then parsed into their own query, with the `*` selector that does not specify a single unbound variable anymore, but shows all variables and their bindings in the result. It looks like Listing 4.2 for our sample query.

```
SELECT * WHERE {
  ?director <http://dbpedia.org/ontology/nationality>
  <http://dbpedia.org/resource/Italy>
};
```

Listing 4.2: Example SPARQL Query

Then, each subquery is mapped to their unbound variables so that we know which variables appear in which subqueries. Unless noted otherwise, the following line numbers reference Algorithm 1, which shows in pseudo-code the client side of the algorithm. Of the set of variables (here: `?director`, `?film`, `?genre`, `?x`), the first one is picked (line 2). Then we iterate over the set of queries mapped to this variable (line 4). The subquery is sent with the picked variable, any previous filters for that variable and a desired exactness measure (that will be explained later) to the servlet. The servlet is on the same virtual server as the SPARQL endpoint, and sends back a Bloom filter with all results

for the picked variable (line 6). This filter is then used for the other subqueries that also search for the picked variable (line 7). Only the results that were already present in the previous result sets are allowed to build the new Bloom filter that will be sent back. This process until now simulates the natural JOIN of the results of the queries in one variable.
When all subqueries for the first picked variable have been processed, the next variable is picked and the belonging subquery set is handled in a similar manner. However, when the subquery has two unbound variables of which one variable has already been handled in the algorithm, its final computed filter is also used for the filter of the new variable. This means for our algorithm, that we send a previous filter of the variable and an additional variable to our servlet (line 5 and 6). In our example query, that would happen if first `?director` got processed and then `?film` is the next variable. In the subquery of the triple pattern `?film <http://dbpedia.org/ontology/director> ?director` both variables appear. So, the result set has one binding for each variable; one for `?film` and one for `?director`. The `?film` binding can only be added to the new Bloom filter if two conditions hold: Firstly, the binding for `?film` appears in any previous filter made by a subquery (Algorithm 2 line 7). Secondly, the binding for `?director` is represented in the resulting Bloom filter of the `?director` cycle (Algorithm 2 line 6). In our example case, the filter for `?film` is still empty, so nothing will be filtered for that variable.
This procedure represents the JOIN of results with multiple (in our case two) overlapping variables. At the end of the iteration over the variables, the resulting Bloom filter cardinality is saved as the approximation value (line 10). In pseudo code, the process after the mapping of the Variables to their subqueries looks the following way:

```
1  Client approximateQuery(q, map, prevFilters, fpp)
2      forall Variable v of Query q do
3          S = map.getSubqueriesOf(v)
4          forall Subquery s in S do
5              additionalVar = (s.listOfVars() without v);
6              Filter bf = adaptFilter(s, v, additionalVar, prevFilters, fpp);
7              prevFilters.put (v, bf);
8          end
9      end
10     return cardinality of prevFilters.last();
```

Algorithm 4.1: Pseudo-Code of algorithm on the client

This approach was constructed in such a way that a Bloom filter has a certain correctness, the FPP. This can be sent to the Servlet, which then hands the argument over to the BloomFilter constructor. The FPP is the probability that an element that is not contained in the set still gets not filtered out when doing the membership test with the filter.

```
1  Servlet adaptFilters(s, v, additionalVar, prevFilters, fpp)
2      ResultList r = Endpoint.executeSelectQuery(s);
3      List acceptedBindings;
4      a = additionalVar;
5      forall ResultBinding b of ResultList r do
6          Boolean cond1 = b.getBindingFor(a) is in prevFilters.get(a);
7          Boolean cond2 = b.getBindingFor(v) is in prevFilters.get(v);
8          if cond1 and cond2 are true then
9              acceptedBindings.add(b.getBindingFor(v));
10     end
11     return createBloomFilter(acceptedBindings, fpp);
```

Algorithm 4.2: Pseudo-Code of algorithm on the servlet

4.3 Expectations

In this section of the chapter I will present the expectations on how the chosen approach compares to the exact calculation of a query result. We want to compare the methods with regard to the runtime and the relative result difference (i.e., the error the approximation produces). The expectations for the experiments are presented as hypotheses, to give structure to the experiments and the following evaluation of the results.

Hypothesis 1. Under normal conditions the approximation should be faster than calculation in some cases; but overall it should not be much slower. This would be the optimal outcome, because the approximation does serve the goal of speeding up queries. This effect should be seen more clearly in a network with high traffic, where the load of the Bloom filter is very light in comparison to the result set for the actual query execution.

Hypothesis 2. The query approximation should be less error prone with a lower chosen FPP. The relative error to FPP relation is expected to start at a relative high level. The curve should drop significantly at the beginning of lowering the FPP, up to a specific FPP value and then practically stagnate. This expectation is made because it is assumed that a Bloom filter gets more correct with a lower FPP, i.e. it will be less likely to overestimate the sub-query result. However, with a tiny FPP it is expected that one filter on its own does not get any more correct. The resulting error should then come mostly from the join of the filters which should then stay constant.

Hypothesis 3. Similar to the error curve, the approximation time in relation to the decreasing FPP should also drop. This is because a lot of false positive results get filtered out and don't have to be joined anymore. However, with a smaller FPP the Bloom filter size increases and testing for membership could affect the runtime more negatively. So it is assumed that after a critical FPP optimum, the time curve should then rise again to represent the overhead of the filter that is chosen too big.

5

Implementation

This chapter will give an overview on the practical implementation done in the scope of this thesis. Section 5.1 presents the structure of the Java code implementation of the program written for the experiments. Section 5.2. will cover a short introduction on the experiments that were done. In Section 5.3. the setup for the experiments is described in more detail.

5.1 Java Client / Server Implementation

The Query approximation and execution itself consists of a program written in Java. It is built in a client / server architecture. For input of the client program, the queries to be run are stored in a textfile. There is also an input file for the address of the webserver the SPARQL endpoint and the servlet run on. For correct query execution, the client part of the application uses *OpenRDF /Sesame* to create a `SPARQLRepository`. To this repository it sends the `TupleQuery` that the input h transformed to. As a result, it gets back a `TupleQueryResult`, that can be processed like an iterator through the resulting `BindingSets`.
To simulate a distributed database environment, the AND clauses of the query are wrapped into SERVICE clauses. This causes the SPARQL endpoint to evaluate each clause at its own, send the subquery to the endpoint that URL that is specified in the clause, and then join the result sets. In our example, the service URL is the same everywhere, but it still forces the chosen SPARQL endpoint to follow the before explained behaviour. This detour was chosen because the effort to handle distributed databases would be too big and is out of the scope of the thesis. It is not one of the goals of this paper to find out how a SPARQL query can be split up onto databases that might answer that query. The whole process of correct query results gets more complicated when datasets on the databases are overlapping, and the merging of the result into a correct answer is a complex process. Still, we wanted to take into consideration that subqueries could be answered from different databases, so we chose to use the SERVICE clauses. Furthermore, if just one big query without SERVICE clauses was sent to the Blazegraph endpoint it could be optimized internally which would again bring an advantage that does not happen in the case with distributed databases.

The same reasoning can be applied as to why not just a COUNT query was sent; the resulting count answers of the subqueries could not be merged into an exact answer if the databases were distributed; so only the resulting answer data was counted. The servlet code is packed into a .war file by *Apache Maven*, and then deployed on the same *Apache Tomcat* as the endpoint web application is run on. It accesses the data storage file of the SPARQL endpoint application and queries this data directly instead of using the application interface. The objects that are sent between the client and servlet are `BloomFilterRequestBeans` and `BloomFilterResponseBeans` serialized by JSON/Jackson. The implementation that was used for the *BloomFilter* class is from Google Guava (the repository can be found on Github [The Google Guava Authors, 2011]).

5.2 Experiments

The experiments are conducted to get results that can either confirm or reject the expectations made above. In the experiments, we run a total of 20 different queries. The chosen queries can be seen in the appendix. For the scope of this thesis, simple SELECT queries with a chain flow of AND clauses were chosen. At first, the exact calculation is run. The execution time and the result number are measured and then saved into an Excel file. This is done to have a comparison to the approximation data.
For the approximation, the queries get run with different FPP values as input. Each of the 20 queries is run with 10 FPP values, ranging from 10^{-2} to 10^{-11}. The runtime of the approximation gets measured again and is saved into an Excel file for each query, together with the approximation result. With this data we can later analyse the three expectation points; overall runtime performance, relative error with respect to the FPP and runtime with respect to the FPP.

5.3 Setup

For the experiments, an environment was setup with the thought in mind to find out the desired comparison aspects. Some aspects were left as near to real world situations as possible, some aspects could be neglected for our evaluation. For the database and database interface we chose to use a Blazegraph endpoint. It was set up as a web application on an Apache Tomcat Servlet container (version 8.0.21). The datasets loaded into the Blazegraph engine were the following:

- **ChEBI** (Chemical Entities of Biological Interest), which contains Linked Data about chemical substances and connections between them (see *https://www.ebi.ac.uk/chebi/faqForward.do*).
- **DrugBank**, a Linked Data collection containing detailed information about drugs and drug target data (see *http://www.drugbank.ca/*).

- **Geonames**, a database that encompasses countries and other geographical places (see *http://www.geonames.org/*).
- **Jamendo**, a database containing Creative Commons licensed music (see *http://dbtune.org/jamendo/*).
- **KEGG** is the Kyoto Encyclopedia of Genes and Genomes (see *http://www.genome.jp/kegg/kegg1.html*).
- **LinkedMDB**, the Linked Movie Database (see *http://www.linkedmdb.org/*).
- **NYTimes**, a database containing carious categories of news vocabulary, including people, organizations and locations (see *http://data.nytimes.com/*, *https://datahub.io/de/dataset/nytimes-linked-open-data*).
- **Semantic Web Dog Food**, a dataset containing information from main conferences and workshops about the Semantic Web (see *http://data.semanticweb.org/*).
- **DBPedia**, a dataset resulting of turning Wikipedia dumps into a structured format (see *http://wiki.dbpedia.org/*).

These datasets were all loaded into the same Blazegraph database journal. The data was chosen with the goal to have a diverse and relatively big database for experiments, to imitate the real Semantic Web. As mentioned before, the distributed data approach was abandoned for practical reasons. The servlet was a .war package and had to be deployed on the local Tomcat server. It is important to note that the namespace 'data' was used to reference the data in the Blazegraph. The client was run locally, with input parameters such as a queries file, a file containing the server-URLs (in our experiments, just the one URL). The experiments were run on two different machines; the first one is a desktop PC (***Run A*** on ***machine A***), the second one is a Lenovo T420 laptop (***Run B*** on ***machine B***). The technical details of these machines can be found in Table 5.1.

Machine	Processors	RAM	Operating System
A	Intel Core i7-3770K @ 4 x 3.5GHz	16 GB	Windows 10 (x64)
B	Intel Core i5-2520M @ 2 x 2.5GHz	8 GB	Windows 8.1 (x64)

Table 5.1: Machines used for the experiments

The resulting data from the experiment (i.e. the runtime and approximation result values) was stored into Excel files. For the analysis, the most interesting statistical property was the relative error. This data was analysed manually and with the help of MatLab R2015b (x64). MatLab was further used to create figures included in the thesis.

6

Evaluation

In this chapter, there will be a presentation and discussion of the results that were retrieved with the before mentioned implementation of the Bloom filter approximation. Following that, there will be a discussion of the limitations that come with these results, and some ideas for future work that come from these limitations.

6.1 Results

This section will be structured after the hypotheses stated in Section 4.3. Only the important results are included in the text to make the reading experience more comfortable. The queries are numbered but do not have any specific ordering. The following statements made and figures shown were made from data extracted of ***Run A***, unless stated otherwise.
Overall it can be said that the resulting data does not show a smooth, regular pattern and is therefore ambiguous to interpret. The query approximation was run with different FPP inputs to determine if this would affect the execution time and result. The data shows that this is indeed the case; lowering the FPP generally had more of a positive impact on both the result exactness and the runtime. The reason for this is that each time a filter is used to create a new one, the error already existing in the first filter can only grow. So if there are less wrong elements getting through the Bloom filter, which can be achieved by lowering the FPP value, this effect cascades and moves further through all following filters. However, as this also affects the size of the filter, it could be speculated that the runtime would grow. The data suggests that this was not the case, in the contrary; the runtime generally sped up with a lower FPP. A possible explanation for this is that there are less elements that will be accepted into the filter. This seemingly cancels out the growing effect of the filter size that the lower FPP leads to.

6.1.1 Hypothesis 1

The first hypothesis stated that the approximation runtime should be faster than the query execution in some cases, overall it should not be slower by a lot. This expectation has not been fully confirmed. The approximation was faster or as fast for only seven of the 20 queries (with a starting FPP of 10^{-3}), as can be seen in Figure 6.1.

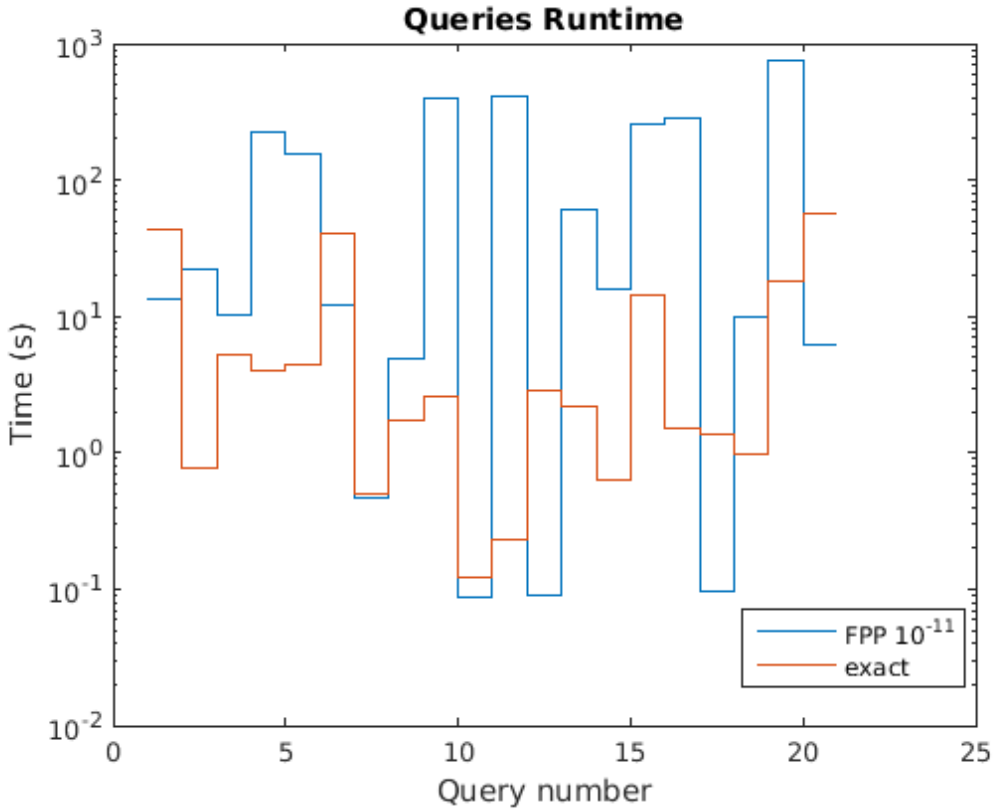

Figure 6.1: Runtime for FPP value of 10^{-11}

Even more, there were queries where the approximation performed far worse time wise than actual execution. It could be assumed that the grammatical structure of these queries was not appropriate for this kind of query approximation. Another cause of the runtime inferiority of the Bloom filter approach could be the use of an index on the data. According to the Blazegraph wiki, it "*is great at fast evaluation of selective queries. A selective query is one where an index can be used to rapidly recover exact data on which the query needs to operate*" [Blazegraph by SYSTAP, LLC, 2015]. Although the Blazegraph endpoint could not have used any optimization for joining the queries, it could be faster in just retrieving the result sets for the subqueries. As explained, in

the approximation servlet the endpoint was not used. The data was directly recovered from the Blazegraph journal, which could have made a difference.
Furthermore, the approximation approach was implemented in standard Java, without any optimization done on the algorithm itself. A more sophisticated algorithm or optimized code implementation (maybe with a more resource-friendly coding language than Java) can surely shorten the approximation time used. Additionally, the computing power and RAM of the devices the servlet and client program run on might play a role for the slower computation of the approximation. While the computing time for the query execution only sped up about 3% maximum on machine A in comparison, for the approximation the differences were bigger (see Figure 6.2).

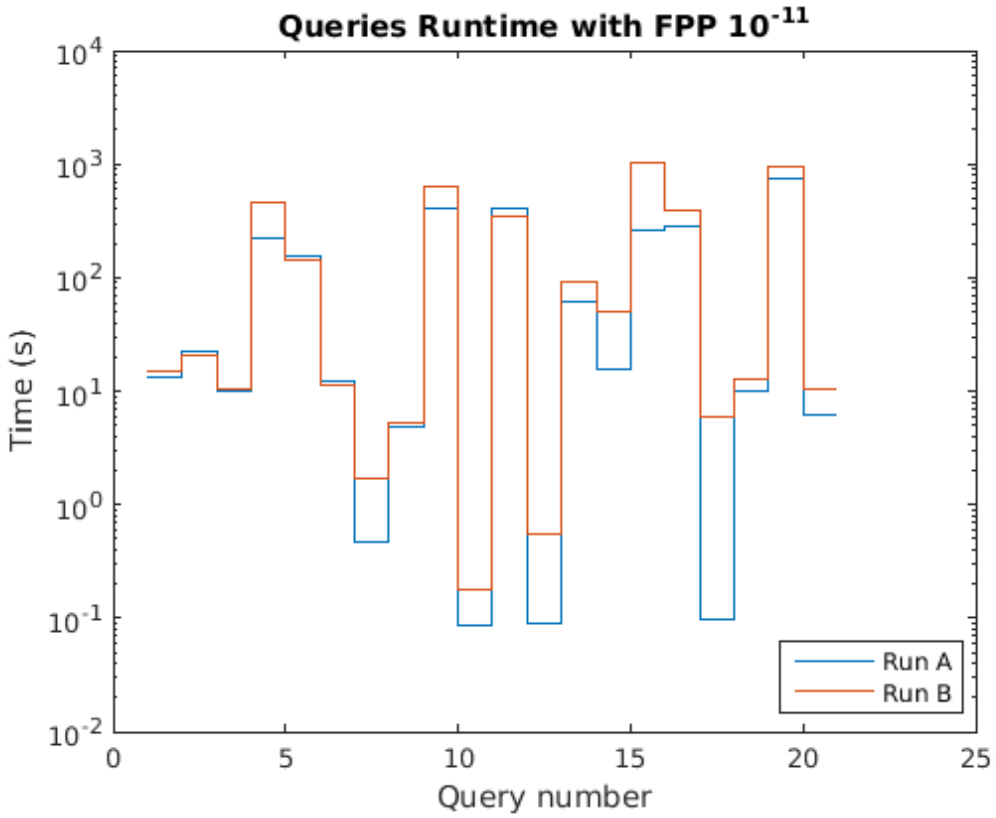

Figure 6.2: Runtime for machine A and B with FPP value of 10^{-11}

Most likely, this is linked to the implementation of the approximation being not very resource-friendly. The approximation process hit a limit while using the allocated RAM and thus spent more time swapping memory on the less powerful machine B. If the servlet was on a dedicated server (still on the same one as the database), as our scenario planned for, the difference could be even more visible.

6.1.2 Hypothesis 2

The second hypothesis makes a statement about the relative error behaviour with different FPP values. Generally, the relative error did not uniformly decrease with a lower FPP. In the figures and tables with the data below one can see that in some cases, the relative error shows an unsteady behaviour. That means, the relative error gets smaller with lower FPP, but then grows again with even lower FPP or does generally show some outlier behaviour. This behaviour was not expected and does not correspond to Hypothesis 2. The error curve according to our expectations should have been a smooth curve starting relatively high, dropping fast at the beginning and stagnating or only falling very slowly after some point. An explanation for this unsteady behaviour could be found when considering the paper "On the false-positive rate of Bloom filters" [Bose et al., 2008]. It states and proves that the FPP equation Bloom derived (see Equation 3.2) is wrong. It is proven that with given values for $\boldsymbol{k \geq 2}$ ***hash functions***, $\boldsymbol{m}$ ***bits*** filter size and $\boldsymbol{n}$ ***elements*** in the filter, the FPP is strictly greater than the formula of Bloom gives. They derive the following equation:

$$p^k < p_{k,n,m} \leq p^k \times \left(1 + O\left(\frac{k}{p}\sqrt{\frac{\ln m - k \ln p}{m}}\right)\right) \tag{6.1}$$

In the Equation 6.1, $p_{k,n,m}$ is the true FPP and p^k the probability that Bloom has derived. However, the Guava implementation code comments state that it uses the formula Bloom derived for calculating the probability as seen the Github repository [The Google Guava Authors, 2011]. So, we do not get results for the exact input FPP we gave, but for a higher value. It seems plausible that some runs with seemingly lower FPP give back a worse result, as the actual FPP must be higher than our input.
An example of that unexpected result was the phenomenon that the relative result error was overall not the best with the lowest FPP. While the experiments with FPP 10^{-2} generally had the worst results (see Table 6.1), the lowering of the FPP did not have a linear effect on the relative error. This can be seen well in Figure 6.3. If we take a look at the relative error average we had over all queries with the FPP chosen as 10^{-11}, it comes down to approximately 30%. Averaging over all 20 queries, the approach with an FPP of 10^{-9} (see Table 6.2) has the best relative error with 12.8%. It is interesting to see how the approximation method is more effective for some queries than others. This can be seen in Figure 6.3 again, where all the queries are shown with their relative error for three different FPP values. Furthermore, the relative error through all FPP values for one single query can have a wide span. In the same Figure, we can see that with query 9 the FPP of 10^{-11} performs with a relative error as low as 0.344% while the biggest relative error is 495.737% with the FPP of 10^{-2}.
The estimating error was not only positive; some results have been underestimated. It may seem counter-intuitive that some results are approximated lower than the actual value. For the over-estimation the main reason is the behaviour of the Bloom filter that allows false positives. With the joins of the different filters, the error gets transferred and can grow. In the test run, there were three queries that were either underestimated or approximated correctly. They are the before mentioned, steady queries.

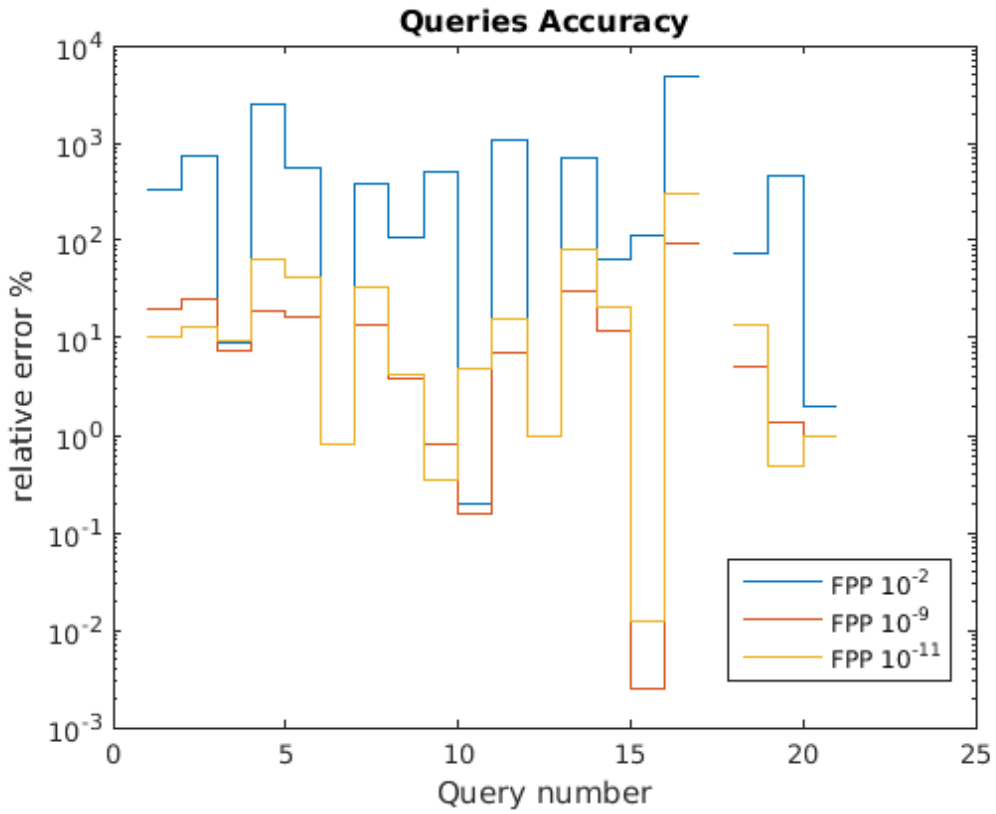

Figure 6.3: Accuracy for FPP values of 10^{-2}, 10^{-9} and 10^{-11}

One explanation for the under-estimation is the following: The Bloom filter in this implementation can only represent a set. It does not change when one tries to insert a variable binding twice. It does not trace the number of elements entered of the same value. This can be a problem when one the variable that we enter into the filter appears in several result triples. It only gets entered into the Bloom filter once and thus is in fact, under-represented. This can be carried onto successive filters.
Here is an example: The first variable, let's say `?a`, has already been handled and the filter has been created. The next variable, `?b`, has included in its first subquery the following resulting triples: (`b1 <knows> a1`), (`b1 <knows> a2`), (`b1 <knows> a3`). In this example `a1`, `a2` and `a3` are values that are a member of `?a` s Bloom filter. This would lead to the input of `b1` into the filter for `?b`. Theoretically, it would get entered three times. However, since our filter does not count the occurrence of the same values, in the end there is only one result for `b1` entered. In such a way or similar, the error multiplies. Of course, especially for the resulting Bloom filter, where we count the cardinality, this becomes the final source of the error.

Query (#)	Approximation time (s)	Result (# of tuples)	Relative error (%)
1	601.765	670	334
2	32.607	721	720
3	64.710	20	9
4	475.960	27211	2472.727
5	169.569	556	555
6	15.0581	1608	0.822
7	2.601	1161	386
8	9.626	3032	107.285
9	581.721	152995	495.737
10	0.118	221	0.194
11	401.008	169432	1064.610
12	0.175	1	0.98
13	64.161	20156	718.857
14	34.160	2503	63.179
15	411.529	135148	110.324
16	333.031	105321	4786.318
17	0.360	1	0
18	13.087	223	73.333
19	776.845	175686	466.25
20	5.399	1152	1.931

Table 6.1: Results for the 20 queries and FPP value of 10^{-2}

Query (#)	Approximation time (s)	Result (# of tuples)	Relative error (%)
1	14.857	41	19.5
2	25.609	26	25
3	11.701	17	7.5
4	209.090	222	19.181
5	156.788	17	16
6	12.085	1598	0.823
7	0.369	44	13.666
8	4.816	134	3.785
9	389.047	558	0.811
10	0.075	214	0.156
11	389.708	1288	7.100
12	0.116	1	0.98
13	62.965	885	30.607
14	16.242	496	11.717
15	253.828	1211	0.002
16	286.810	2037	91.590
17	0.107	1	0
18	12.992	18	5
19	769.266	878	1.335
20	9.101	0 28	1

Table 6.2: Results for the 20 queries and FPP value of 10^{-9}

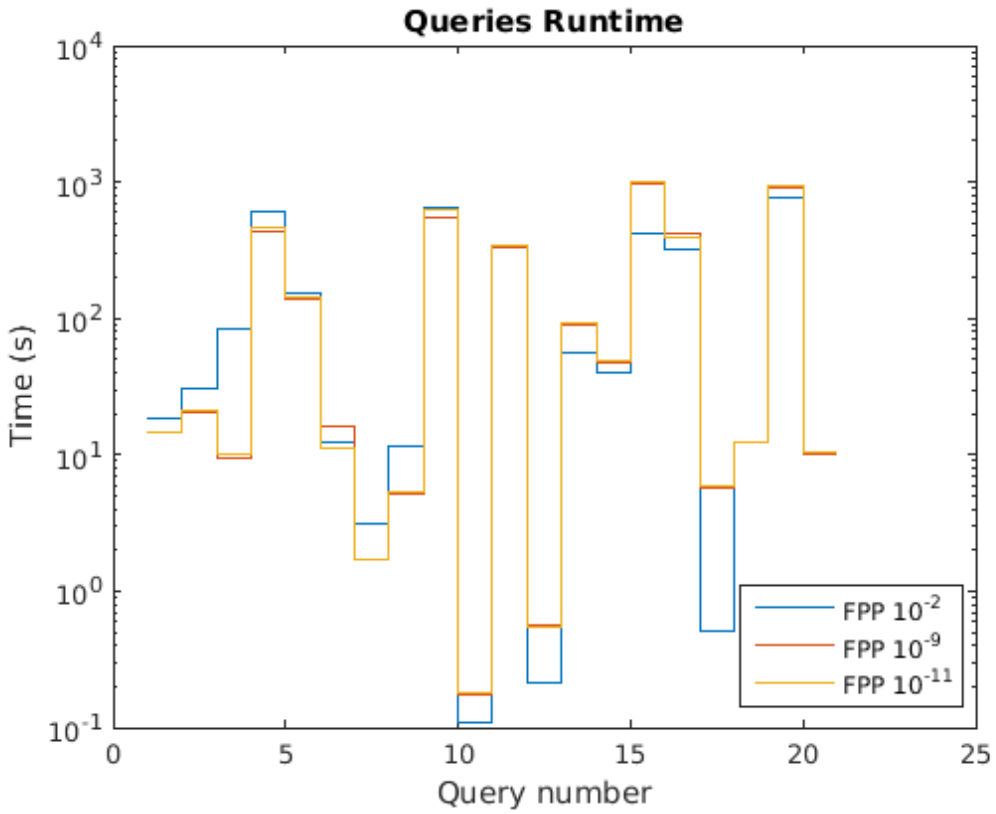

Figure 6.4: Runtime for FPP values of 10^{-2}, 10^{-9} and 10^{-11}

6.1.3 Hypothesis 3

Finally, in the third hypothesis the expectation was formulated that the approximation runtime should decrease when lowering the FPP value. What is quite surprising is the little effect the lowering of the FPP had. Over the different values, there are of course differences; sometimes they are quite major (see Figure 6.4 with data from ***Run B***) but mostly they are not. However, in average the FPPs all performed badly on the same queries. Along the lines of what was expected, some FPPs also performed better than the calculation on the same queries. Between the exactness levels in these specific cases there were only minor differences in the runtime, with the exception of the FPP 10^{-2} which delivers again results that are mostly worse than the others. It seems this probability value is too low for both the results and the approximation time to be comparable to the others. With this, Hypothesis 3 can be partly confirmed. While lowering the FPP value had some positive effect on the runtime, a distinct, optimal value where the runtime started to rise again could not be found. Instead, the time seemed to stay near the same value or oscillate nearly to that value with higher FPPs.

In summary, it can be said that 10^{-2} is the one FPP that delivered the worst results regarding relative error. Lowering the FPP has mostly positive effects on the relative error; the overall most positive results were achieved with an FPP of 10^{-9}.
If we take a look at the relative errors the approximation resulted in, we see that most queries had a tolerable percentage, with a relative error mean of 12.79% and a median of 6.05% for the best performing FPP. There are some outliers; but what is more important is that different FPP performed the best for different queries. Since it can't be guessed which FPP yields the best outcome for a query, it is advised to choose 10^{-9}, which had overall the best results. Or it can be argued that either 10^{-8} or 10^{-10} should used if the relative error is not as important as the time constraint, which was in average slightly better for those two over all queries. However, there were generally no extreme differences in the runtime of FPP values higher than 10^{-2}.

6.2 Limitations and Future Work

The experiments in this thesis were conducted trying to simulate real life conditions. However, that was not possible in some aspects that will be elaborated here. One example for this is the distributed data. With both the Bloom filter method and the conventional computation of the query we could handle distributed data. It would be possible to handle the joins at the client side in the Bloom filter case. With the exact computation, the usage of the service clause makes it possible to send the query to a single endpoint where the subqueries are then sent further and at last joined again.

However, one problem that was omitted in this thesis is the mapping of subqueries to data endpoints. If the datasets in different endpoints did not overlap at all and the subqueries referenced only one dataset, the mapping could be done with a single ask query for each subquery. The formulation of the previous statements already highlights possible complications. If the datasets overlap, or subqueries refer to multiple endpoints the mapping is more complicated. This is a possibly frequent situation in reality, since one of the main advantages of Linked Data is that datasets from different sources can be combined to find out new information. While this is something that would be interesting to solve and compare different methods for, it is not a subject this thesis tries to examine. For the sake of comparing the query execution, we assume the mapping has already been done and use the combined datasets as our endpoint. Since the mapping would pose the same problem for actual query execution as for query approximation, it would not cause any time or result difference between the two approaches.

What was further done was a manual optimization of the queries. The AND clauses were ordered in such a way that our algorithm could cope with them. For the Bloom filter joins to work, the variables have to be ordered in such a way that a connection exists between them. This connection consists of a subquery where the two unbound variables take the place of the aubject and object. This means that our queries have ordered AND clauses, so that they link the variables similar to a chain through the whole query. This

is one adaption to the queries that should be possible to solve in a programmatic way. Again, this was omitted and the simple, manual solution was chosen. The optimization was applied to the queries before they were entered as input, so the query execution had the same conditions as the query approximation.

The Bloom filter implementation of this thesis is the standard one that has some restrictions other variants of Bloom filter do not have. Firstly, it can only represent a set, which does not include multiple identical values for the same variable. This introduces an underestimation error, of which an explanation is contained in the results section. This additional error could maybe be solved with another implementation of the filter, which is called the Counting Bloom Filter (e.g. as mentioned in **Network Applications of Bloom Filters: A Survey** [Broder and Mitzenmacher, 2003]). Secondly, when using our chosen Bloom filter implementation, we have to know how many elements we want to insert into it to guarantee a specific FPP. This means that we need to make a list of all definite members of the candidates returned by the query, and go through the list again to insert the data into the filter. This introduces an overhead into the algorithm that can be considered unnecessary. A Bloom filter approach that is worth mentioning in this context is the dynamic Bloom filter mentioned in **Theory and Network Applications of Dynamic Bloom Filters** [Guo et al., 2006]. It works without knowing in advance how many elements will be inserted into the filter. A comparison between multiple variants of a Bloom filter used in a similar implementation could be a further input for future work.

A further limitation of this paper concerns the setup of the experiment, that were done locally and on a standard PC and a Lenovo T420 laptop. To gain further insights and verify the results statistically, the experiments have to be run a considerable amount of time. The implementation used in the experiments uses a simple, single-threaded method so all queries are executed in succession. That means the experiments took a long time to complete and were thus small in number. A future parallel implementation (e.g. on a machine cluster) would speed up the whole experiments greatly, and would facilitate a more thorough testing with a more adequate number of runs. Moreover, it would be interesting to see how the Bloom filter approach would compete in a true server setting with a high network load. One of the main advantages of the method is that Bloom filters are considerably smaller to send through the net. In the setup of the experiments, that advantage could not be fully exploited. In a distributed setup of the data endpoints, they would have to send tuples to each other to be joined. In the current setup this all happens on the same server, which causes the correct query execution a slight advantage.

7

Conclusions

This chapter will conclude the thesis. It presents the most important findings of this thesis and includes a summary of the results section.
The aim of this thesis was to review different approaches for Query approximation, analyse them for suitability with RDF datasets and SPARQL queries and then implement and evaluate a solution in Java code. From three different approaches, the Bloom filter approach was chosen for efficiency reasons - the transformation of the method to an algorithm for the SPARQL and RDF domain was seen as the least time-costly. Furthermore, the broad usage of Bloom filters in different applications speaks for a well-tested behaviour suitable for several use cases. All in all, the Bloom filter promised a good cost to benefit ratio for the approximate query processing with the given time restraints.
The derived algorithm uses Bloom filters to represent the results of subqueries and also to serve as a join mechanism with its filtering of unsuitable elements. With the implementation in the java code, some constraints had to be made, for example the distributed databases could not be modelled exactly. An example for this is that the datasets had to be loaded into a single SPARQL endpoint. With some detours, like the SERVICE clauses, the experiment environment was modelled slightly more realistic. However, these adaptions have to be kept in mind when interpreting the results.
The results themselves are partly ambiguous as to the whole usefulness of the presented query approximation approach. Especially Hypothesis 1 could not generally be confirmed, the approximation outperformed the execution only for less than half of the test queries. Some queries were processed a lot slower with the approximation, which leads to the conclusion that the chosen approach is not useable for a general speedup of querying in this format. However, some assumptions as to the factors that could have caused the slowness were made, including the naïve implementation of the algorithm in Java, the devices that the experiments were run on (especially that client and server run on the same machine) and the grammatical structure of some queries. These assumptions could be tested with further work.
Hypothesis 2 could be partly confirmed. There was one FPP that stood out as the best approach with respect to the average relative error over all 20 queries. Although that would indicate an optimum FPP value for our algorithm, the behaviour of the relative error was not as continuous as expected. The data revealed quite an unsteady effect on the lowering of the FPP value. Instead of a continuously decreasing error we found irregular up and downs on the relative error. An explanation offered refers to the calculation

of the FPP used in the code implementation of the Bloom filter. The implementation used an approach to calculate the number of hash functions and the bit size of the filter from the FPP and element insertion input, that differs from the correct mathematical equation. Additionally to this phenomenon, the general overestimation and underestimation of the results have been explained in the results section. Using other Bloom filter implementations seems to offer practical solutions to reduce the estimation error.
Hypothesis 3 could be confirmed in the aspect that generally, the runtime in our experiments decreased with a lower FPP value. Mostly, the time difference was not major between the values. We found that a probability of 10^{-8} or 10^{-10} delivered slightly faster results averaged over all queries. In conclusion, there has been no clear indication for an optimum FPP value for the query runtime with the FPP range that was tested. An approach for further work would be to expand the FPP value range and enlarge the space between the values.
The resulting data suggests a neutral conclusion on the usage of Bloom filters for query approximation with the approach we have taken in this thesis. However, when paying attention to the limitations and the ideas for improvement of the approach there are some optimizations still to be made. They might deliver a more promising result when being implemented in future work.

References

[Agarwal et al., 2014] Agarwal, S., Milner, H., Kleiner, A., Talwalkar, A., Jordan, M., Madden, S., Mozafari, B., and Stoica, I. (2014). Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 481–492, New York, NY, USA. ACM.

[Antonopoulos, 2014] Antonopoulos, A. M. (2014). *Bloom Filters*, chapter 6: The Bitcoin Network, pages 154–159. O'Reilly Media.

[Berners-Lee, 2006] Berners-Lee, T. (2006). Design issues: Linked data. https://www.w3.org/DesignIssues/LinkedData.html. Online; last accessed in June 2016.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, 284:29–37.

[Blazegraph by SYSTAP, LLC, 2015] Blazegraph by SYSTAP, LLC (2015). Optimizations and benchmarking: Query optimization. https://wiki.blazegraph.com/wiki/index.php/QueryOptimization. Online; last accessed in June 2016.

[Blazegraph by SYSTAP, LLC, 2016] Blazegraph by SYSTAP, LLC (2016). Product. https://www.blazegraph.com/product/. Online; last accessed in June 2016.

[Bloom, 1970] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

[Bose et al., 2008] Bose, P., Guo, H., Kranakis, E., Maheshwari, A., Morin, P., Morrison, J., Smid, M., and Tang, Y. (2008). On the false-positive rate of Bloom filters. *Inf. Process. Lett.*, 108(4):210–213.

[Broder and Mitzenmacher, 2003] Broder, A. and Mitzenmacher, M. (2003). Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509.

[Chang et al., 2006] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2006). Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on*

Operating Systems Design and Implementation, OSDI '06, pages 205–218, Berkeley, CA, USA. USENIX Association.

[Corby et al., 2006] Corby, O., Dieng-Kuntz, R., Gandon, F., and Faron-Zucker, C. (2006). Searching the Semantic Web: approximate query processing based on ontologies. *IEEE Intelligent Systems*, 21(1):20–27.

[Dell'Era, 2007] Dell'Era, A. (2007). Join over histograms. http://www.adellera.it/investigations/join_over_histograms. Online; last accessed in June 2016.

[Eclipse RDF4J, 2016] Eclipse RDF4J (2016). Welcome to rdf4j. http://rdf4j.org/. Online; last accessed in June 2016.

[Guéret et al., 2008] Guéret, C., Oren, E., Schlobach, S., and Schut, M. (2008). An evolutionary perspective on approximate rdf query answering. In *Proceedings of the 2Nd International Conference on Scalable Uncertainty Management*, SUM '08, pages 215–228. Springer Berlin Heidelberg.

[Guo et al., 2006] Guo, D., Wu, J., Chen, H., and Luo, X. (2006). Theory and network applications of dynamic bloom filters. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–12.

[Hashem et al., 2015] Hashem, I. A. T., Yaqoob, I., Anuar, N. B., Mokhtar, S., Gani, A., and Khan, S. U. (2015). The rise of "big data" on cloud computing: Review and open research issues. *Inf. Syst.*, 47:98–115.

[Hausenblas et al., 2008] Hausenblas, M., Halb, W., Raimond, Y., and Heath, T. (2008). What is the size of the Semantic Web. In *In Proceedings of the International Conference on Semantic Systems (ISemantics) 2008.*

[Hecht, 2011] Hecht, Fabio V; Bocek, T. S. B. (2011). B-tracker: Improving load balancing and efficiency in distributed p2p trackers. In *Proceedings of the International Conference on Peer-to-Peer Computing*, pages 310–313. IEEE Computer Society.

[Hilbert and López, 2011] Hilbert, M. and López, P. (2011). The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65.

[Huang et al., 2008] Huang, H., Liu, C., and Zhou, X. (2008). Computing relaxed answers on RDF databases. In *Web Information Systems Engineering - WISE 2008: 9th International Conference, Auckland, New Zealand, September 1-3, 2008. Proceedings*, pages 163–175, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Jamard and Gardarin, 2007] Jamard, C. and Gardarin, G. (2007). *Extending an XML Mediator with Text Query*, pages 111–124. Springer Berlin Heidelberg.

[Mitzenmacher, 2001] Mitzenmacher, M. (2001). Compressed Bloom Filters. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 144–150, New York, NY, USA. ACM.

[Muralikrishna and DeWitt, 1988] Muralikrishna, M. and DeWitt, D. J. (1988). Equi-depth multidimensional histograms. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 28–36, New York, NY, USA. ACM.

[Papapetrou et al., 2010] Papapetrou, O., Siberski, W., and Nejdl, W. (2010). Cardinality estimation and dynamic length adaptation for Bloom Filters. *Distrib. Parallel Databases*, 28(2-3):119–156.

[Reddy and Kumar, 2010] Reddy, B. R. K. and Kumar, P. S. (2010). Efficient approximate sparql querying of web of linked data. In *Proceedings of the 6th International Conference on Uncertainty Reasoning for the Semantic Web - Volume 654*, URSW'10, pages 37–48, Aachen, Germany, Germany. CEUR-WS.org.

[The Apache Software Foundation, 2016a] The Apache Software Foundation (2016a). Apache tomcat. http://tomcat.apache.org/. Online; last accessed in June 2016.

[The Apache Software Foundation, 2016b] The Apache Software Foundation (2016b). Welcome to apache maven. https://maven.apache.org/. Online; last accessed in June 2016.

[The Google Guava Authors, 2011] The Google Guava Authors (2011). Guava bloomfilter and bloomfilterstrategies. https://github.com/google/guava/tree/master/guava/src/com/google/common/hash. Online; last accessed in June 2016.

[W3C, 2016] W3C (2016). About w3c. https://www.w3.org/Consortium/. Online; last accessed in June 2016.

[W3C Recommendation, 2008] W3C Recommendation (2008). SPARQL query language for RDF. https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. Online; last accessed in June 2016.

[Yakunin, 2010] Yakunin, A. (2010). Nice bloom filter application. http://blog.alexyakunin.com/2010/03/nice-bloom-filter-application.html. Online; last accessed in June 2016.

A

Appendix

A.1 Setting up the Experiments

First of all, the Java code has to be set up. For this, one can choose to either use the .jar and .war files included on the CD directly (found in the code folder). For direct use, the bloomfilterclient.one-jar.jar file is suggested. It can be run directly on the terminal with the command `java -jar bloomfilterclient.one-jar.jar`. But first, the Tomcat server should be set up with the web applications and it should be running.
Alternatively, the code can be run with Maven to produce an own .jar / .war file. This way, you can choose alternative inputs for the program to run with. The file for the query input (named 'Queries.txt') can be found in the BloomFilterClient under the directory src/main/resources. Placed in the same folder is the input file for the location of the SPARQL Endpoints, 'ServerURIs.txt'. On the servlet, there is a file names blazegraph.properties, found in the folder src/main/resources. With the parameter `com.bigdata.journal.AbstractJournal.file=location/bigdata.jnl` we can define the location of the data we want to access. This location is later relative to the /bin path in the Tomcat environment. In the working example, this data is a copy of the one the Blazegraph web application uses, placed into a new folder with name data.
For the setup of the Servlet and Blazegraph, first you have to install an Apache Tomcat distribution and download the Blazegraph web application (.war file). The Tomcat has to be unpacked and set up in the usual way. It has to be paid attention to the fact that the port under /conf/server.xml that listens to the HTTP should be the same as the port used in the ServerURIs.txt input file. It is defined with the tag: `<Connector port="portnumber" protocol="HTTP/1.1" connectionTimeout=.../>`.
To use the web application on the Tomcat, the .war files have to be placed into the /webapps folder. They are automatically unpacked for usage when the Tomcat is started. For a Windows system, navigate on the terminal into the /bin folder and run startup.bat. In the code that was supplied, the data file for the Servlet access has been specified to be /bin/data/bigdata.jnl. This was done to avoid any locks or access denied errors. The file there is a copy of the /bin/bigdata.jnl file.
To fill the database, you can download data dumps from the websites that are presented in the Section 5.3 Setup, or choose your own data dumps. The data format we chose was .nt files, because they can be read into the Blazegraph easily. For this, just enter

the absolute location of the .nt file into the Blazegraph Update tab. Before this, you should create and use a new Namespace with the name data and Mode triples.
After starting the Tomcat Server, the Servlet should be running and ready to access.

A.2 Glossary

Apache Maven "Apache Maven is a software project management and comprehension tool" [The Apache Software Foundation, 2016b]. It was used to compile Java projects with their required library files.

Apache Tomcat This is the name of an "open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies" [The Apache Software Foundation, 2016a]. It is used to deploy web applications and act as a Webserver.

Approximate Query Processing (AQP) Generally stands for an approach to estimate the result of a database query. It generally trades correctness of the result for faster execution time.

Big Data The term that is used "to refer to the increase in the volume of data that are difficult to store, process, and analyze through traditional database technologies" [Hashem et al., 2015].

Blazegraph "An ultra-scalable, high-performance graph database with support for the Blueprints and RDF/SPARQL APIs" [Blazegraph by SYSTAP, LLC, 2016]. Was used in the experiments for storing the RDF datasets and querying it with SPARQL.

Bloom Filter A space-efficient structure to represent a data set and test whether elements are included in the stored set or not.

FPP False positive probability. Here: The probability that an element is not in the data set but the membership test with the corresponding Bloom filter returns that it is a member.

Linked Data A term that specifies the usage of RDF to publish and connect information from different sources in the Semantic Web.

Ontology A formal model that defines characteristics of data and relationships between objects from a specific domain.

OpenRDF / Sesame This is a Java framework for handling the querying and the processing of RDF data. It is now called RDF4J. More information on [Eclipse RDF4J, 2016].

Peer to Peer (P2P) A system where all participating nodes are (ideally) equal in their role and function, as opposed to the traditional client / server architecture of systems.

RDF Stands for Resource Description Framework. RDF is a directed, labelled graph data format used for representing information in the Web.

Semantic Web An extension of the known Web constructed to make sharing and searching for data easier for computers.

SPARQL A querying language similar to SQL, but especially built for executing queries on RDF / Semantic Web data [W3C Recommendation, 2008].

URI Uniform Resource Identification. Can be used to identify resources, be it digital objects on the Internet or on a computer, or physical objects.

URL Uniform Resource Locator. Is, for example, used to present addresses of websites in an easy to read format for humans.

World Wide Web Consortium (W3C) A "community that develops open standards to ensure the long-term growth of the web" [W3C, 2016].

XML Extensible Markup Language. A language that can be used to format documents to a state where it's readable for humans as well as for computers.

List of Figures

List of Tables