FACULTY OF ENGINEERING
AND ARCHITECTURE

Scaling networks of personal data stores through Approximate Membership Functions

Thomas Devriese
Student number: 01806904

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. ir. Ruben Verborgh

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2020-2021

GHENT
UNIVERSITY

FACULTY OF ENGINEERING
AND ARCHITECTURE

Scaling networks of personal data stores through Approximate Membership Functions

Thomas Devriese
Student number: 01806904

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. ir. Ruben Verborgh

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2020-2021

GHENT
UNIVERSITY

Acknowledgements

The master's dissertation is an integral part of obtaining a master's degree, and an opportunity to contribute to the academic world. As we live in a society that is digitally transforming at an increasing pace, my research *'Scaling networks of personal data stores through Approximate Membership Functions'* was certainly topical. In combination with my interest in the Web, this motivated me to successfully bring this study to an end.

I would like to thank everyone who helped me reach this goal, starting with my promoter dr. ir. Ruben Taelman. He guided me in every way he could and provided me with his extensive knowledge on the topic. I would also like to give my thanks to my mentor at the university, dr. Marleen Denert, for always answering any questions concerning the master's dissertation and the curriculum in general. Lastly, I would like to thank my parents and the rest of my family, my girlfriend Lisa, and my friends for always supporting me throughout the course of my education.

Thomas Devriese, March 2021

Usage

Scaling networks of personal data stores through Approximate Membership Functions

by
Thomas Devriese

Master's dissertation submitted in order to obtain the academic degree of Master of Science in Information Engineering Technology

Academic year 2020-2021

Ghent University
Faculty of Engineering and Architecture
Department of Information Technology
Chairman: Prof. dr. ir. Bart Dhoedt

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. ir. Ruben Verborgh

Summary

Decentralized social Web applications are advantageous in comparison to their centralized counterparts for various reasons, including the improvement of their users' privacy and the reusability of data across multiple applications. Solid, a decentralized Web ecosystem, makes these applications possible by letting users store their own data in personal online data stores or pods. Users can then decide which applications have access to which parts of their data. However, as user data can be distributed across thousands of remote pods, querying over this data becomes difficult. In this research, we implement source selection based on Approximate Membership Functions in order to reduce the number of HTTP requests to be performed by the query engine. This way, we try to make networks of personal data stores more scalable.

Samenvatting

Gedecentraliseerde sociale webapplicaties tonen verschillende voordelen in vergelijking met hun gecentraliseerde tegenhangers, waaronder de verbetering van de privacy van hun gebruikers en de herbruikbaarheid van data doorheen meerdere applicaties. Solid, een gedecentraliseerd web-ecosysteem, maakt dit soort applicaties mogelijk door gebruikers hun eigen data te laten opslaan in persoonlijke online datastores of pods. Als gevolg kunnen gebruikers beslissen welke applicaties toegang hebben tot welke delen van hun gegevens. Omdat gebruikersdata echter gedistribueerd kan zijn over duizenden externe pods, wordt het opvragen van deze gegevens moeilijk. In dit onderzoek implementeren we bronselectie op basis van Approximate Membership Funtcions om het aantal HTTP-requests dat door de query-engine moet worden uitgevoerd, te verminderen. Op deze manier proberen we netwerken van persoonlijke datastores schaalbaarder te maken.

Keywords: Semantic Web, Linked Data, decentralized social Web applications, source selection

Trefwoorden: Semantisch Web, Linked Data, gedecentraliseerde sociale webapplicaties, bronselectie

Scaling networks of personal data stores through Approximate Membership Functions

Thomas Devriese

Supervisors: dr. ir. Ruben Taelman, prof. dr. ir. Ruben Verborgh

Abstract - **Decentralized social Web applications are advantageous in comparison to their centralized counterparts for various reasons, including the improvement of their users' privacy and the reusability of data across multiple applications. Solid, a decentralized Web ecosystem, makes these applications possible by letting users store their own data in personal online data stores or pods. Users can then decide which applications have access to which parts of their data. However, as user data can be distributed across thousands of remote pods, querying over this data becomes difficult. In this research, we implement source selection based on Approximate Membership Functions in order to reduce the number of HTTP requests to be performed by the query engine. This way, we try to make networks of personal data stores more scalable.**

Keywords - **Semantic Web, Linked Data, decentralized social Web applications, source selection**

I. Introduction

Data on the Web is now more prominent than ever. Using data, we can communicate with each other, find the nearest grocery store, order pizza, and so much more. To enable not only humans, but also machines to actually understand information on the Web, the Semantic Web [1], [2] was introduced. This extension of the Web makes data machine-readable by giving meaning or semantics to it.

In this context, the idea of Linked Data [3] was developed, which encourages people to publish and interlink as much data as possible, following predefined guidelines. The result is a large Web of Data (WoD), which is expanding exponentially. The Linking Open Data (LOD) cloud [4], an initiative that visualizes this WoD, currently contains 1255 datasets interconnected by 16174 links.

This concept of Linked Data opens up a new world of possibilities. One of these possibilities, which has already been brought into reality, is the Social Linked Data (Solid) project. [5] Solid is an ecosystem that defies the traditional method of centralized data storage by providing users with their own personal online data store (POD). As it is based on the principles of Linked Data, information stored in the pods can be linked, creating a network of personal data stores. This enables the realization of decentralized social Web applications, which provide some major advantages in comparison to their centralized counterpart. For one, giving users control over their own data significantly improves their privacy. Secondly, storing a user's data at a single location avoids duplication of data and lets users reuse information across multiple decentralized social Web applications.

However, in the case of large-scale decentralized Web applications, data might be spread across thousands or millions of remote datasources. In contrast to centralized social networks, where all data is fetched from a single database, retrieving data in a large-scale decentralized environment is not that straightforward. To this end, in this research we will try to find a way to make efficient data retrieval in decentralized networks of personal data stores more attainable.

We will rely on the implementation of a source selection process in a client-side query engine to achieve this goal. Source selection reduces the number of datasources to be contacted for a given query, by ruling out irrelevant datasources for that query. More specifically, we will use Approximate Membership Functions, which are small probabilistic data structures, to enable source selection.

The outline of this work is as follows. Section 2 looks into related work. Section 3 and section 4 discuss the implementation and the experimental setup, respectively. The results of the experiments are analyzed in section 5 and further discussed in section 6. Finally, some conclusions are drawn in section 7, followed by a summary of possible future work in section 8.

II. Related Work

A. The Semantic Web

The Semantic Web is an idea which Tim Berners-Lee, who also invented the World Wide Web, came up with. It is an extension of the current Web, in which information can be processed, understood, and manipulated by computers. The idea behind it is to give meaning to data by applying a well-defined structure to it. RDF [6], a language that represents information in the Web, expresses this meaning. It does this by uniquely identifying any object and interlinking these objects to form webs of information. [1]

B. Linked Data

The idea of connecting all related data to form a Web of Data is referred to as Linked Data by Tim Berners-Lee. [3] It is a set of rules for publishing and linking information on the Web. Initially, the Web consisted of linked documents, without machines being able to comprehend the actual meaning of the information. Now, the Web is evolving into a global space in which both documents and data are connected. [7] This upgraded version of the Web has many advantages and applications. Linked Data search engines, for example, have the ability to aggregate data from many data sources by following links between them. The accumulated data can then be queried.

C. The Resource Description Framework

The Resource Description Framework (RDF) presents a data model optimized for linking data that defines resources. This model is graph-based, which means it can be visualized by a directed labeled graph. This makes RDF suitable for representing social network data. [7], [9] A resource is an object that can be identified by a URI. [10] When Hypertext Transfer Protocol (HTTP) URIs are used, these resources can be looked up on the Web. [3]

RDF can encode data as triples. These triples consist of a subject, a predicate (also called a property type) and an object. The subject (a resource) and predicate of a triple are represented by URI's, whereas the object may be a URI or a string. [7] A predicate and an object together form a property of a subject. [8] The relation between a subject and an object is determined by the predicate. A Web of Data is established when for instance a subject from one dataset is linked to an object from another dataset. In this case, the RDF triple becomes an RDF link. Both the subject and the object must then be URI references. [7]

For data to be exchanged and processed, a general syntax or data serialization format is required. [8] Using such a format, an RDF graph can have a textual representation. There are several RDF syntaxes, including RDF/XML, N-Triples, Turtle and JSON-LD. [11]

D. *SPARQL*

Data encoded in the RDF format does not have much use if it cannot be queried in some way. For that matter, the SPARQL Protocol and RDF Query Language (SPARQL) [9], [12] was developed by W3C in 2008. SPARQL is a query language for RDF encoded data and can be used for querying multiple graphs. In addition to the possibility of executing SPARQL queries over RDF dumps, content providers can set up a SPARQL endpoint. A big advantage of these endpoints is that the data does not necessarily need to be stored in RDF, as it can be created at run-time from other data sources.

A SPARQL query consists of a basic graph pattern, which is a series of triple patterns. These are much like RDF triples, but they can contain variables. The subject, the object or the predicate of the triple pattern may be replaced by a variable. When RDF terms (RDF literals, IRIs, or blank nodes) from a subgraph can be replaced by variables, resulting in a graph equal to that subgraph, then the basic graph pattern matches the subgraph. A successful SPARQL query results in a sequence of solutions, specified by the variables present in the `SELECT` clause. This sequence can be represented as a table, with one solution per row. The basic graph pattern is defined in the `WHERE` clause. Note that for every solution, all the variables mentioned in the `WHERE` clause must bind to the RDF terms of the solution.

E. *Solid*

The Social Linked Data (Solid) project [5] puts users in control of their own data. Although referred to as a platform, Solid is in fact a protocol or ecosystem for decentralized social Web applications, working on top of existing W3C recommendations and the Linked Data stack. [13] With Solid, users each have one or more personal online data stores (PODs), in which all of their data is stored. These pods can be accessed through the Web and are independent from applications, which means that the same data can be reused throughout multiple applications. Users decide which applications and which people have access to which parts of their data. This results in a major improvement in protecting people's privacy. [5]

The Solid ecosystem was built on top of some existing W3C standards. It relies on the principles of the Semantic Web, which allows for interlinking data that resides anywhere in the Web. Standards used by Solid include RDF, Linked Data, SPARQL, WebID, Web Access Control, LDP and Linked Data Notifications. [5], [15], [16]

F. *Querying over Linked Data*

The difficulty with decentralized environments using Linked Data is data retrieval. Especially in use cases like decentralized social Web applications, query processing becomes a challenge as the number of independent data sources can increase quickly. Currently there are two main approaches [17], [18] for data integration and querying in a decentralized environment.

The first approach is warehousing, also called materialization-based approaches. This method assumes a single, central database, in which RDF dumps of all remote data sources are collected in advance. By applying preprocessing and indexing techniques on the aggregated data, queries can easily be answered using the central repository. Data warehousing offers the fastest query response times, but it has some major disadvantages. One of these is that the data in the central repository is not always synchronized with its sources.

The second method is distributed query processing (DQP) or federated querying. This approach does not need a central database, but instead queries the data sources directly. To that end, input queries are parsed and split into separate subqueries, which are then sent to the individual data sources, according to source selection. Finally, the responses from the remote sources are combined into a valid result. Distributed query processing is advantageous in various ways. One advantage is that, in contrast to warehousing, the data is always synchronized, as there is only one copy of it. Unfortunately, availability and reliability of this federated approach cannot be guaranteed because the system relies on a lot of possibly unstable or inactive data stores. [18]

G. *Source selection*

Most distributed query processing approaches are not very scalable when the network of data sources gains in size. Existing methods are often designed to handle only a small number of endpoints exposing a fairly large amount of data, instead of thousands of separate sources, each containing fewer resources. The latter case, however, would be ideal for decentralized social Web applications, whereby possibly thousands or millions of users, located all around the world, each have their own personal data pod. But when there is a very large number of data sources, checking every source for potential data contributing to the result of a query is unattainable. To that end, a source selection process must be executed, in order to rule out sources that do not contain valuable data for the query and keep only those that can contribute to the query answer. This process takes a triple pattern as input and returns a set of data sources that contain potential variable bindings. [18]

Two effective data structures to enable source selection over many sources are data summaries [16], [18] and Approximate Membership Functions (AMFs). [16], [20], [21] They are both probabilistic, which means that they can produce false negatives, but they cannot produce false positives. The chance of false positives occurring can be chosen by setting a false positive probability.

AMFs have two different implementations, being Bloom filters [22] and Golomb-coded sets. [23]

III. Implementation

This research applies Approximate Membership Functions to a Solid environment in order to evaluate their effect on query performance. The emphasis lies on scalability of decentralized networks of datasources, by simulating an expanding social network. With this study, we hope to get somewhat closer to making large-scale decentralized social Web applications reality.

A. *Data generation*

In order to simulate a social network to run experiments over, a lot of synthetic data had to be generated. For this, a decentralized version of the LDBC SNB Data Generator [24] was used.

The decentralized version splits the large Turtle file, created by the SNB data generator, so that the data is spread across a multitude of files, each containing the data of one specific entity in the social network. More specifically, each file contains all the triples of which the subject is one specific URI, denoting an entity in the network. In this research, for the purpose of performance, only the files containing data of persons and cities were used, with a maximum of 3500 persons and the 1231 cities they live in.

B. *Server*

Behind the scenes, the decentralized data generator uses the Community Solid Server (CSS) [25] to host and serve the RDF files over HTTP. CSS is an open-source and modular implementation of the Solid specifications. Using the server, developers can create decentralized Solid applications and experiment with them. The decentralized data generator tool exploits the file-based store functionality of the Solid Server to serve the generated social network data over HTTP. That way, a decentralized social network can be simulated, as the data of every single entity must be accessed using a separate URI.

C. *Approximate Membership Functions*

For the construction of the Approximate Membership Functions, a separate script was written. The script uses Bloom filters as the probabilistic data structure. First, the number of bits in the bitmap of the filter and the number of hash functions are determined, based on the number of triples in the RDF file and the desired false positive probability. Then, a Bloom filter is initialized for each term (subject, predicate, object). Finally, the corresponding term of each triple is added to the filter. The pseudocode below shows the AMF construction algorithm.

```
m = calculateBitmapSize()
k = calculateNumberOfHashes()
filters = ['subject','predicate','object']
for (variable in filters):
  filters[variable] = new Bloem(m, k)
for (triple in triples):
  for (variable in filters):
    filters[variable].add(Buffer.from(triple[variable]))
for (variable in filters):
  filters[variable] = {
    type: 'http://semweb.mmlab.be/ns/membership#BloomFilter',
    filter: filters[variable].bitfield.buffer.toString('base64'),
    m: m,
    k: k
  }
```

D. *Client-side query engine*

The most important aspect of the implementation of this research was the process of extending a client-side query engine to support source selection based on Approximate Membership Functions. For this, we used the Comunica [26] query engine. Comunica is a modular Web-based SPARQL query engine that allows for the development and testing of new Linked Data query processing functionalities. It enables federated querying over heterogeneous datasource interfaces out-of-the-box.

For this research, the modularity of Comunica was exploited to implement source selection capabilities based on AMFs. A new actor was developed for the sole purpose of filtering the datasources array.

E. *Source selection*

The purpose of the generated Approximate Membership Functions is to enable a source selection process, in which many of the irrelevant datasources for a given query can be filtered out. This leaves a smaller set of sources to be queried over, which leads to better performance. The lower the chosen false positive probability for the AMFs, the smaller the filtered set of sources. The pseudocode below shows the algorithm used for source selection.

```
filteredSources = []
for source in originalSources:
  addSource = true
  for term in ['subject','predicate','object']:
    if triplePattern[term] not variable:
      if not source.bloomFilter[term].contains(triplePattern[term]):
        addSource = false
  if addSource:
    filteredSources.push(source)
```

IV. Experimental Setup

To ensure the results to be as reliable as possible, the experiments were run with many different combinations of parameters. First, 15 different SPARQL queries were tested. Next, the number of datasources in the social network was varied to obtain a network of 10, 100, 500, 1000, 2000 and 3500 sources. Thirdly, six different false positive probabilities were tested for the Bloom filters, being 1/4096, 1/1024, 1/128, 1/64, 1/4 and 1/2. Finally, each possible combination of the aforementioned parameters was iterated three times. Each time, the average of the metrics of those three iterations was calculated to obtain the definitive result.

In addition to the number of results produced by the query, four other metrics were measured, namely the number of HTTP requests issued by the query engine, the total query execution time, the client-side memory usage, and the client-side CPU load. Unfortunately, the CPU load metric produced seemingly random values, making it unusable in the discussion of the experiment results.

In order to make the experiment reproducible, a Bash script was written, which iteratively invokes the query engine with different parameters. This way, the whole experiment could be run by executing only one script. A query timeout was set for 5 minutes, in combination with a maximum memory usage limit of 4096 MB for Nodejs. All experiments were run on a single machine with an Intel Core i7-8705G CPU at 3.10 GHz and 8 GB of RAM.

V. Results

A. *HTTP requests*

The first and most important metric to evaluate is the number of HTTP requests performed by the querying process. Figure 1 shows the number of HTTP requests per number of datasources, averaged over all queries (lower is better). The legend shows the different false positive probabilities of the Bloom filters, whereby *Default* refers to the use of the query engine without Approximate Membership Functions. The default query engine will from now on be referred to as E, while the AMF-extended query engine will be called E'.

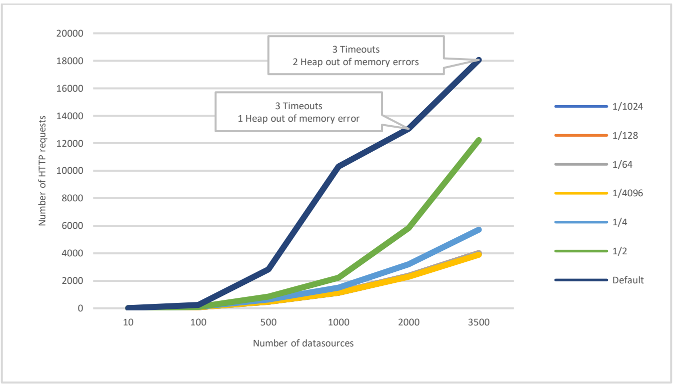


Figure 1 - Average number of HTTP requests per number of datasources.

Note that the chart is not completely accurate, as several queries (executed over 2000 and 3500 datasources) resulted in a timeout or a *heap out of memory* error. This only occurred in the default version of the query engine, not with the use of AMFs. Due to these failures, no metrics have been measured and as such, the number of HTTP requests of these queries has not been included in the averages. Therefore, in reality, the averages for 2000 and 3500 datasources using the default version are much higher, following the pattern found in the lower numbers of datasources.

Still, the default version of the query engine immediately stands out, as the number of HTTP requests increases much quicker than with the use of AMFs. Going from 500 to 1000 datasources in the default version results in an increase of 264%, while the increase with the use of AMFs at a false positive rate $p = 1/64$ is just 137%. The difference in the average number of HTTP requests between E and E' (at $p = 1/64$) for 500 datasources is 2347 requests. The same difference for 1000 datasources is 9152 requests, which means an increase of 290%.

While the number of HTTP requests at $p = 1/4096$, $p = 1/1024$, $p = 1/128$ and $p = 1/64$ almost perfectly align, a significant increase is observed at rates of $p = 1/4$ and $p = 1/2$. At 3500 datasources, $p = 1/2$ performs more than 3 times as many HTTP requests in comparison to $p = 1/4096$. Therefore, high false positive probabilities contribute to poorer query performance.

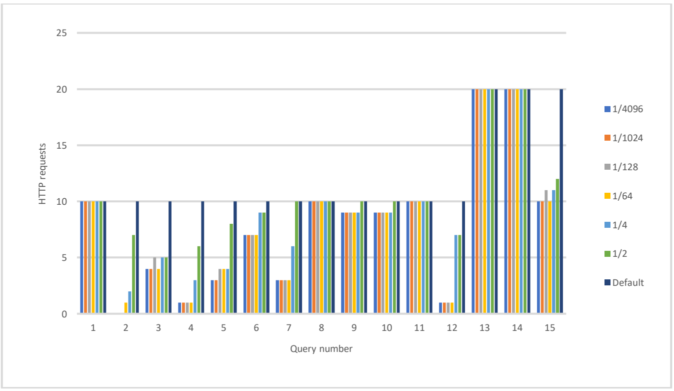


Figure 2 - Number of HTTP requests per query for 10 datasources.

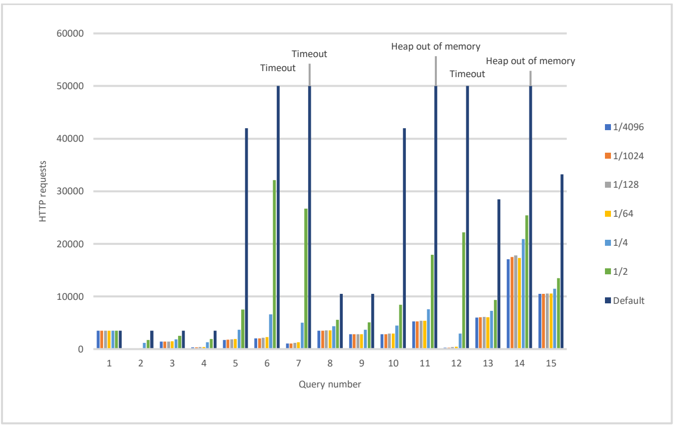


Figure 3 - Number of HTTP requests per query for 3500 datasources.

The above clustered column charts were constructed to examine the results more into detail. These histograms show the exact number of HTTP requests for each executed query (averaged over 3 iterations). Figure 2 and Figure 3 show the observations for 10 and 3500 datasources, respectively. It is clear that an increase in the number of datasources leads to a bigger difference between E and E', at least for low false positive rates.

B. *Query execution time*

In addition to the number of HTTP requests, the total query response times were also measured. This metric is directly affected by the HTTP requests, as an important part of the querying process involves contacting remote datasources. Figure 4 illustrates the total query execution time per number of datasources, averaged over all queries (lower is better).

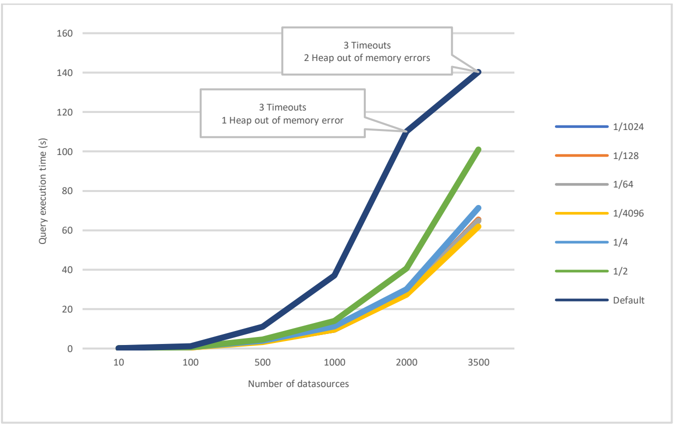


Figure 4 - Average query execution time per number of datasources.

The same pattern arises as with the number of HTTP requests. E' with low false positive probabilities shows the best response times, followed by E' with higher false positive rates. The averages of E are relatively close to those of E' for very small networks but increase much more rapidly when the network expands. Going from 500 to 1000 datasources while using E' and $p = 1/64$ produces an increase of 162% in the average query response time. This increase rises to 236% using the default query engine. The difference between E and E' (at $p = 1/64$) when scaling up from 500 to 1000 sources shows an increase of 273%.

False positive rates $p = 1/4$ and $p = 1/2$ again show reduced performance in comparison to lower rates. $p = 1/4096$ contributes to slightly better results than $p = 1/1024$,

$p = 1/128$ and $p = 1/64$, of which the averages almost perfectly align.

The clustered column charts are left out here, as the results are the same as with the number of HTTP requests: the difference between E and E' also grows with the expansion of the network of datasources.

C. *Memory usage*

The third measured metric is the memory usage at the client, who runs the query engine. Just as the query response time, memory usage is also greatly affected by the number of HTTP requests. More sources to be retrieved means more data to keep in memory. Figure 5 displays the evolution of the memory usage with an increasing number of sources, averaged over all queries (lower is better).

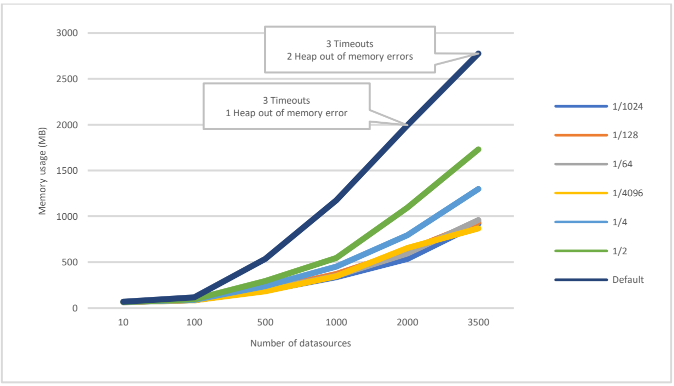


Figure 5 - Average memory usage per number of datasources.

As the number of HTTP requests has a big influence on the client memory usage, we can observe the same pattern here. Just as with the use of AMFs, the memory usage of the regular query engine setup starts off fairly low. But again, adding more datasources means a faster increase for E than for E'. Moving from 500 to 1000 datasources using the default engine and with $p = 1/64$ causes an increase of 78% in the average memory usage at the client. The same calculation for the default engine yields an increase of 119%. The difference between E and E' (at $p = 1/64$) when scaling up from 500 to 1000 data stores increases 143%.

Lower false positive rates again show better performance than $p = 1/4$ and $p = 1/2$. Oddly, the probability $p = 1/4096$ causes higher memory usage for 2000 datasources than $p = 1/1024$, $p = 1/128$ and $p = 1/64$, but then again lower usage for 3500 sources. This is likely due to the lower accuracy of the memory usage metric.

Again, the clustered column charts are left out, as the same pattern can be observed as with the previous two metrics.

D. *AMF construction*

To evaluate the performance of the AMF construction process, some experiments were carried out in which the time it takes to create the Bloom filters was measured, together with the disk space these filters require. As the experiments in the previous sections were executed over a maximum of 3500 datasources, each containing one person's info, the same 3500 sources are used in the experiments in this section. However, an additional 1231 datasources, each containing the data of a location, are included as well. This brings the total to 4731 datasources for the experiments concerning AMF construction.

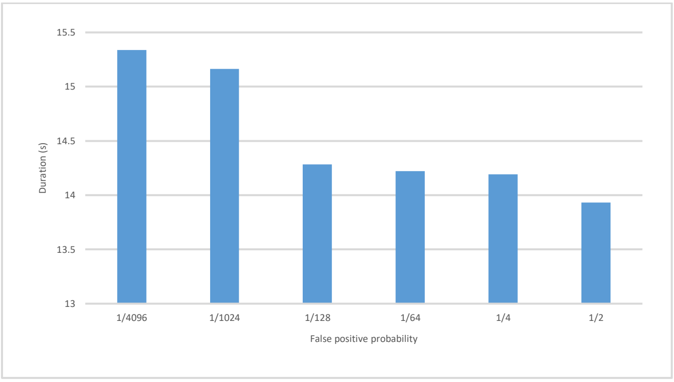


Figure 6 - Construction time per false positive probability for the Bloom filters of 4731 datasources.

Figure 6 clarifies that a lower false positive probability leads to a higher overall construction time, with a maximum of 15.3 seconds for $p = 1/4096$ (approximately 3 milliseconds per datasource). This behavior is expected, as a lower false positive rate means more precision, and so the filter must contain more data and thus, it takes more time to create. The difference between the two extremes is relatively small, being only 1.4 seconds. However, this number can quickly rise in networks with a scale of hundreds of thousands or even millions of datasources.

If we look at the disk space these Bloom filters require, the same pattern can be observed. This is illustrated in Figure 7. Adding more precision to the Bloom filters, requires them to contain more data and take up more space. Although the difference in construction time between the lowest and highest false positive rate is relatively small, the difference in size between these two extremes is a lot bigger. The filters with a probability of $p = 1/4096$ require almost 3 times as much space as the filters with $p = 1/2$. In this setting, that difference is only 2.37 MB, but in networks with very large scale, the difference can be crucial.

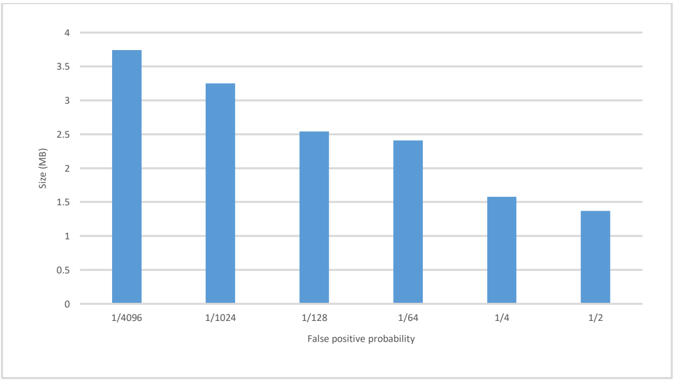


Figure 7 - Disk space per false positive probability for the Bloom filters of 4731 datasources.

In the previous sections, we learned that lower false positive rates do it better, performance-wise. Contrarily, in this section we discovered that Bloom filters with lower probabilities take more time to create and require more disk space. Therefore, the optimal balance between query performance, construction time and disk space lies somewhere between the highest and the lowest false positive probability.

It seems $p = 1/64$ would be the recommended false positive rate, as it performs much better than $p = 1/4$ and $p = 1/2$, while less time and disk space are required to create the filters than with $p = 1/4096$, $p = 1/1024$ and $p = 1/128$ (and it barely differs from them performance-wise).

VI. Discussion

The use of Approximate Membership Functions when querying over an expanding network of datasources proves to be advantageous in multiple ways. The number of HTTP requests, as well as the overall query execution time and the client-side memory usage are improved significantly by using Bloom filters. We saw that the difference between the regular query engine and the AMF-extended query engine is negligible for very small networks with no more than 100 datasources, but as soon as that number increases, the difference increases with it.

However, the problem with the current method of applying Approximate Membership Functions is that, without the use of an external server taking care of the AMFs, the number of HTTP requests at the client can rise up to more than the number of HTTP requests when using the default query engine. This is because of the need of the Bloom filters to be constantly up to date. Given an input query, the client would first have to contact every datasource in the network in order to construct the filters. Only thereafter would the client be able to perform source selection and execute the query. As such, the total number of HTTP requests for each query would be equal to the sum of the number of datasources in the network and the number of sources retrieved from the source selection process. As a result, query performance would be worse than with the default setup.

Luckily, there are a few solutions to this problem, of which two will be discussed here. Both of them implement an aggregator, which is separated from the client. Both also have their own preferable use cases.

The first solution is to add an aggregator to the network, which crawls from datasource to datasource and keeps a list of all sources in the network. It can renew all AMFs periodically, or it may generate a new filter upon file changes, in which case the datasource can send a notification to the aggregator. When the client wants to perform a query, it can let the aggregator know, after which the aggregator combines all filters into one large file and sends it back to the client. Then, the client can perform source selection and execute the query. This method adds only one HTTP request to the complete process of the client. Moreover, this solution becomes particularly interesting if the client needs to execute multiple queries across the same range of sources. In that case, it can download the combined AMF file from the aggregator once and reuse it for different queries. However, this approach is not ideal for very large networks of sources, as the file containing the AMFs grows with the number of sources and must be sent to the client over the network. For $p = 1/64$ and 4731 sources, sending a file with a size of 2.41 MB is achievable. Scaling the network up to 100.000 sources, however, leads to a size of 51 MB, while a network of 1 million sources means an unattainable 510 MB.

The second method is more favorable to large-scaled networks. In this approach, the aggregator discovers new sources and updates its AMFs in the same way, but it differs in the querying process itself. When the client must execute a query, it first sends the query over to the aggregator. After that, the aggregator performs the source selection process itself, and sends the list of selected sources back to the client. Based on this list of sources, the client can immediately execute the query. Just as with the first solution, this approach adds only one HTTP request to the complete querying process of the client. In this method, the size of the network matters less, as the query and the list of sources are the only pieces of data to be exchanged with the aggregator. However, this solution demands more processing power from the aggregator. Furthermore, this method may be less ideal in situations where protection of privacy plays a very important role, as each query from the client must be sent over the network and can be read by the aggregator. Therefore, the choice of which solution to implement mostly depends on the use case, but this trade-off needs to be further investigated in future work.

VII. Conclusions

In this research, we tried to find a way to make querying over a large number of datasources more feasible. By investigating this matter, large-scale decentralized social Web applications could become reality. The proposed method is to extend the client-side query engine by implementing source selection based on Approximate Membership Functions. By performing source selection, irrelevant datasources for a given input query can be filtered out, significantly reducing the number of HTTP requests to be executed by the client. Approximate Membership Functions can be given a false positive probability. A higher probability means a higher chance for an irrelevant source to end up in the selected list of sources. There is however a trade-off, as AMFs with a lower false positive probability are bigger in size.

Some experiments were set up, in order to measure the query performance of a query engine extended with AMFs (E'), in comparison to the default engine (E). The goal was to find that the difference in performance between E' and E increased with the expansion of the network of datasources. For this, three different metrics were measured: the number of HTTP requests, the total query execution time, and the client-side memory usage.

For all three metrics, we saw the same pattern. In very small networks, the metrics of E' and E are almost equal. However, the more datasources are added, the more the difference in performance between E' and E grows, with E' providing much better results than E. Some queries in the regular setup even failed due to timeouts or *heap out of memory* errors, while these queries caused no issues in the AMF setup.

Different false positive probabilities were also tested, in order to find the optimal balance between query performance, filter construction time and filter size. We discovered that a false positive rate of $p = 1/64$ proved to contribute to the best results overall.

Although the use of Approximate Membership Functions provides a significant improvement in query performance, the construction of these filters must also be considered. In a setup where the client itself must maintain the AMFs, the total number of HTTP requests invoked by a query is much higher than in a regular setup without AMFs. This causes the query response times to escalate significantly. However, two solutions have been proposed to this problem, both involving an external aggregator.

Both of these methods add only one HTTP request to the complete querying process of the client, making the complete process still very performant in comparison to the regular query engine setup. However, the approaches differ in other ways and each have their own preferable use cases.

To summarize, we can state that the use of Approximate Membership Functions when querying over a large number of sources provides a major improvement to query performance, but only if the resources are available to implement an external aggregator, which manages the AMFs. Unfortunately, even with the use of AMFs, the measured metrics are for some queries over many sources still rather poor to make responsive decentralized social Web applications possible. Nonetheless, the findings in this research might help us take a step closer to that goal.

VIII. Future Research

Various subjects may be further investigated to continue the work of this research. In the discussion, we mentioned two solutions for reducing the number of HTTP requests using the proposed AMF-enabled query engine in combination with an aggregator. Although the optimal use cases and the advantages and disadvantages of each approach were briefly discussed, these methods must be further investigated and experimented with in future research, as the correct use of aggregators in a decentralized environment can significantly improve query performance.

Furthermore, other AMF approaches should be tested, such as compressed Bloom filters and Golomb-coded sets. These data structures are more space-efficient than regular Bloom filters. Therefore, they could be particularly useful in the first of the two mentioned solutions regarding aggregators, as the biggest issue there is that sending a large AMF file over the network drastically reduces query performance.

In fact, Approximate Membership Functions are not the only way of implementing source selection in a client-side query engine in the context of Solid. Other summarization approaches must be investigated and tested as well.

Finally, decentralized social Web environments could be simulated in larger scale, and queried over using more elaborate queries. We have a long way to go before we can achieve truly large-scale decentralized social Web applications, but the first steps have already been taken.

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Sci. Am.*, 2001.

[2] N. Shadbolt, W. Hall, and T. Berners-Lee, "The Semantic Web Revisited," *IEEE Intell. Syst.*, vol. 21, no. 3, pp. 96–101, 2006, doi: 10.1109/MIS.2006.62.

[3] T. Berners-Lee, "Linked Data," Jul. 27, 2006. https://www.w3.org/DesignIssues/LinkedData.html (accessed Oct. 15, 2020).

[4] J. P. McCrae *et al.*, "The Linked Open Data Cloud," May 20, 2020. https://lod-cloud.net/ (accessed Mar. 23, 2021).

[5] A. V. Sambra *et al.*, "Solid: A Platform for Decentralized Social Applications Based on Linked Data," 2016. Accessed: Oct. 08, 2020. [Online]. Available: https://diasporafoundation.org.

[6] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," Feb. 25, 2014. https://www.w3.org/TR/rdf11-concepts/ (accessed Oct. 21, 2020).

[7] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - The story so far," *Int. J. Semant. Web Inf. Syst.*, vol. 5, no. 3, pp. 1–22, 2009, doi: 10.4018/jswis.2009081901.

[8] E. Miller, "An Introduction to the Resource Description Framework," *Bull. Am. Soc. Inf. Sci. Technol.*, vol. 25, no. 1, pp. 15–19, Jan. 2005, doi: 10.1002/bult.105.

[9] S. Harris, A. Seaborne, and E. Prud'hommeaux, "SPARQL 1.1 Query Language," Mar. 21, 2013. https://www.w3.org/TR/sparql11-query/ (accessed Oct. 24, 2020).

[10] J. Van Ossenbruggen, L. Hardman, and L. Rutledge, "Hypermedia and the Semantic Web: A Research Agenda," 2001.

[11] G. Schreiber and Y. Raimond, "RDF 1.1 Primer," Jun. 24, 2014. https://www.w3.org/TR/rdf11-primer/ (accessed Oct. 20, 2020).

[12] B. Quilitz and U. Leser, "Querying distributed RDF data sources with SPARQL," *Lect. Notes Comput. Sci.*, vol. 5021 LNCS, pp. 524–538, 2008, doi: 10.1007/978-3-540-68234-9_39.

[13] R. Buyle *et al.*, "Streamlining governmental processes by putting citizens in control of their personal data," 2019.

[14] A. Sambra, A. Guy, S. Capadisli, and N. Greco, "Building Decentralized Applications for the Social Web," in *Proceedings of the 25th International Conference Companion on World Wide Web - WWW '16 Companion*, 2016, pp. 1033–1034, doi: 10.1145/2872518.2891060.

[15] T. Berners-Lee and R. Verborgh, "Solid: Linked Data for personal data management," Oct. 08, 2018. https://rubenverborgh.github.io/Solid-DeSemWeb-2018/ (accessed Nov. 10, 2020).

[16] R. Taelman, S. Steyskal, and S. Kirrane, "Towards Querying in Decentralized Environments with Privacy-Preserving Aggregation," 2020.

[17] P. Haase, T. Mathäß, and M. Ziller, "An evaluation of approaches to federated query processing over linked data," *ACM Int. Conf. Proceeding Ser.*, no. January, 2010, doi: 10.1145/1839707.1839713.

[18] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over Linked Data," *World Wide Web*, vol. 14, no. 5, pp. 495–544, 2011, doi: 10.1007/s11280-010-0107-z.

[19] H. Stuckenschmidt, R. Vdovjak, G. J. Houben, and J. Broekstra, "Index structures and algorithms for querying distributed RDF repositories," *Thirteen. Int. World Wide Web Conf. Proceedings, WWW2004*, pp. 631–639, 2004, doi: 10.1145/988672.988758.

[20] M. Vander Sande, R. Verborgh, J. Van Herwegen, E. Mannens, and R. Van de Walle, "Opportunistic linked data querying through approximate membership metadata," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9366, no. Iswc, pp. 92–110, 2015, doi: 10.1007/978-3-319-25007-6_6.

[21] R. Taelman, J. van Herwegen, M. Vander Sande, and R. Verborgh, "Optimizing approximate membership metadata in triple pattern fragments for clients and servers," *CEUR Workshop Proc.*, vol. 2757, pp. 1–16, 2020.

[22] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970, Accessed: Mar. 22, 2021. [Online]. Available: https://doi.org/10.1145/362686.362692.

[23] F. Putze, P. Sanders, and J. Singler, "Cache-, Hash-, and space-efficient bloom filters," *ACM J. Exp. Algorithmics*, vol. 14, Sep. 2008, doi: 10.1145/1498698.1594230.

[24] R. Taelman, "rubensworks/ldbc-snb-decentralized: A tool to create a decentralized version of the LDBC SNB social network dataset, and serve it over HTTP.," 2021. https://github.com/rubensworks/ldbc-snb-decentralized (accessed Mar. 19, 2021).

[25] R. Verborgh, J. Van Herwegen, and R. Taelman, "solid/community-server: Community Solid Server: an open and modular implementation of the Solid specifications." https://github.com/solid/community-server (accessed Mar. 20, 2021).

[26] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, "Comunica: a Modular SPARQL Query Engine for the Web," *Proc. 17th Int. Semant. Web Conf.*, vol. 11137, pp. 239–255, 2018.

Netwerken van persoonlijke datastores schaalbaar maken aan de hand van Approximate Membership Functions

Thomas Devriese

Promotoren: dr. ir. Ruben Taelman, prof. dr. ir. Ruben Verborgh

***Abstract* - Gedecentraliseerde sociale webapplicaties tonen verschillende voordelen in vergelijking met hun gecentraliseerde tegenhangers, waaronder de verbetering van de privacy van hun gebruikers en de herbruikbaarheid van data doorheen meerdere applicaties. Solid, een gedecentraliseerd web-ecosysteem, maakt dit soort applicaties mogelijk door gebruikers hun eigen data te laten opslaan in persoonlijke online datastores of pods. Als gevolg kunnen gebruikers beslissen welke applicaties toegang hebben tot welke delen van hun gegevens. Omdat gebruikersdata echter gedistribueerd kan zijn over duizenden externe pods, wordt het opvragen van deze gegevens moeilijk. In dit onderzoek implementeren we bronselectie op basis van Approximate Membership Funtcions om het aantal HTTP-requests dat door de query-engine moet worden uitgevoerd, te verminderen. Op deze manier proberen we netwerken van persoonlijke datastores schaalbaarder te maken.**

***Trefwoorden* - Semantisch Web, Linked Data, gedecentraliseerde sociale webapplicaties, bronselectie**

I. Introductie

Data op het web is nu belangrijker dan ooit. Met behulp van data kunnen we met elkaar communiceren, de dichtstbijzijnde supermarkt vinden, pizza bestellen, en nog veel meer. Om niet alleen mensen, maar ook machines in staat te stellen informatie op het web daadwerkelijk te begrijpen, werd het Semantisch Web [1], [2] geïntroduceerd. Deze uitbreiding van het Web maakt gegevens leesbaar voor machines door er betekenis of *semantiek* aan te geven.

In dit verband is het idee van Linked Data [3] ontwikkeld, dat mensen aanmoedigt om zoveel mogelijk gegevens te publiceren en aan elkaar te koppelen, volgens vastgestelde richtlijnen. Het resultaat is een groot Web of Data (WoD), dat zich exponentieel uitbreidt. De Linking Open Data (LOD) cloud [4], een initiatief dat deze WoD visualiseert, bevat momenteel 1255 datasets die onderling verbonden zijn door 16174 links.

Dit concept van Linked Data opent een nieuwe wereld van mogelijkheden. Eén van deze mogelijkheden, die reeds in de praktijk is gebracht, is het Social Linked Data (Solid) project. [5] Solid is een ecosysteem dat de traditionele methode van gecentraliseerde gegevensopslag tegengaat door gebruikers hun eigen persoonlijke online datastore (POD) te bieden. Aangezien het gebaseerd is op de principes van Linked Data, kan informatie opgeslagen in de pods aan elkaar worden gekoppeld, waardoor een netwerk van persoonlijke datastores ontstaat. Dit maakt de realisatie mogelijk van gedecentraliseerde sociale webapplicaties, die een aantal grote voordelen bieden in vergelijking met hun gecentraliseerde tegenhangers. Door gebruikers controle te geven over hun eigen gegevens wordt hun privacy aanzienlijk verbeterd. Ten tweede voorkomt het opslaan van de gegevens van een gebruiker op een enkele locatie duplicatie van gegevens en stelt het gebruikers in staat informatie te hergebruiken doorheen meerdere gedecentraliseerde sociale webapplicaties.

In het geval van grootschalige gedecentraliseerde applicaties kunnen gegevens echter verspreid zijn over duizenden of miljoenen externe gegevensbronnen. In tegenstelling tot gecentraliseerde sociale netwerken, waar alle gegevens uit één enkele database worden gehaald, is het ophalen van gegevens in een grootschalige gedecentraliseerde omgeving niet zo eenvoudig. Daarom zullen we in dit onderzoek proberen een manier te vinden om het efficiënt ophalen van gegevens in gedecentraliseerde netwerken van persoonlijke datastores haalbaarder te maken.

We zullen een bronselectieproces implementeren in een client-side query engine om dit doel te bereiken. Bronselectie vermindert het aantal databronnen dat gecontacteerd moet worden voor een gegeven query, door irrelevante databronnen voor die query uit te sluiten. Meer specifiek zullen wij gebruik maken van Approximate Membership Functions, kleine probabilistische datastructuren, om bronselectie mogelijk te maken.

De opzet van dit werk is als volgt. Sectie 2 gaat in op verwant werk. Sectie 3 en sectie 4 bespreken respectievelijk de implementatie en de experimentele opzet. De resultaten van de experimenten worden geanalyseerd in sectie 5 en verder besproken in sectie 6. Ten slotte worden enkele conclusies getrokken in sectie 7, gevolgd door een samenvatting van mogelijk toekomstig werk in sectie 8.

II. Gerelateerd Werk

A. *Het Semantisch Web*

Het Semantisch Web is een idee van Tim Berners-Lee, die ook het World Wide Web heeft uitgevonden. Het is een uitbreiding van het huidige Web, waarin informatie door computers kan worden verwerkt, begrepen en gemanipuleerd. Het idee erachter is betekenis te geven aan gegevens door er een vastgelegde structuur op aan te brengen. RDF [6], een taal die informatie op het Web weergeeft, drukt deze betekenis uit. Dit gebeurt door elk object op unieke wijze te identificeren en deze objecten onderling te verbinden om informatiewebben te vormen. [1]

B. *Linked Data*

Het idee om alle verwante gegevens met elkaar te verbinden om zo een Web of Data te vormen, wordt door Tim Berners-

Lee "Linked Data" genoemd. [3] Het is een verzameling regels voor het publiceren en koppelen van informatie op het Web. Aanvankelijk bestond het Web uit gekoppelde documenten, zonder dat machines in staat waren de werkelijke betekenis van de informatie te begrijpen. Nu ontwikkelt het Web zich tot een wereldwijde ruimte waarin zowel documenten als gegevens met elkaar worden verbonden. [7] Deze opgewaardeerde versie van het Web heeft vele voordelen en toepassingen. Zoekmachines voor Linked Data hebben bijvoorbeeld de mogelijkheid om gegevens uit vele gegevensbronnen samen te voegen door links tussen deze bronnen te volgen. Op deze verzamelde gegevens kunnen vervolgens zoekopdrachten worden uitgevoerd.

C. *Het Resource Description Framework*

Het Resource Description Framework (RDF) biedt een datamodel dat is geoptimaliseerd voor het koppelen van data die resources definiëren. Dit model is graaf-gebaseerd, wat betekent dat het kan worden gevisualiseerd door een gerichte gelabelde graaf. Dit maakt RDF geschikt voor het representeren van data voor sociale netwerken. [7], [9] Een resource is een object dat kan worden geïdentificeerd door een URI. [10] Wanneer Hypertext Transfer Protocol (HTTP) URI's worden gebruikt, kunnen deze bronnen op het Web worden opgezocht. [3]

RDF kan gegevens coderen als triples. Deze triples bestaan uit een subject, een predicaat (ook wel een property type genoemd) en een object. Het subject (een resource) en predicaat van een triple worden voorgesteld door URI's, terwijl het object een URI of een string kan zijn. [7] Een predicaat en een object vormen samen een eigenschap van een subject. [8] De relatie tussen een subject en een object wordt bepaald door het predicaat. Een Web of Data komt tot stand wanneer bijvoorbeeld een subject uit een dataset wordt gekoppeld aan een object uit een andere dataset. In dit geval wordt de RDF triple een RDF link. Zowel het subject als het object moeten dan URI-referenties zijn. [7]

Om data te kunnen uitwisselen en verwerken is een algemene syntax of serialisatieformaat nodig. [8] Met behulp van zo'n formaat kan een RDF graaf een tekstuele representatie krijgen. Er zijn verschillende RDF syntaxen, waaronder RDF/XML, N-Triples, Turtle en JSON-LD. [11]

D. *SPARQL*

Data gecodeerd in RDF formaat heeft niet veel nut als het niet op een of andere manier kan worden opgevraagd. Daarom is SPARQL Protocol and RDF Query Language (SPARQL) [9], [12] ontwikkeld door W3C in 2008. SPARQL is een querytaal voor RDF gecodeerde data en kan worden gebruikt voor het opvragen van data over meerdere grafen. Naast de mogelijkheid om SPARQL queries uit te voeren over RDF dumps, kunnen content providers een SPARQL endpoint opzetten. Een groot voordeel van deze endpoints is dat de data niet noodzakelijk in RDF hoeft te zijn opgeslagen, omdat deze at run-time uit andere databronnen kan worden gecreëerd.

Een SPARQL query bestaat uit een basic graph pattern, dat een serie van triple patterns is. Deze lijken veel op RDF triples, maar ze kunnen variabelen bevatten. Het subject, het object of het predicaat van het triple pattern kan worden vervangen door een variabele. Wanneer RDF termen (RDF literals, IRI’s, of blank nodes) uit een subgraaf vervangen kunnen worden door variabelen, wat resulteert in een graaf gelijk aan die subgraaf, dan komt het basic graph pattern overeen met de subgraaf. Een succesvolle SPARQL query resulteert in een opeenvolging van oplossingen, gespecificeerd door de variabelen in de `SELECT`-clausule. Deze reeks kan worden voorgesteld als een tabel, met één oplossing per rij. Het basic graph pattern van de graaf wordt gedefinieerd in de `WHERE`-clausule. Merk op dat voor elke oplossing, alle variabelen vermeld in de `WHERE` clausule moeten binden aan de RDF termen van de oplossing.

E. *Solid*

Het Social Linked Data (Solid) project [5] geeft gebruikers de controle over hun eigen gegevens. Hoewel Solid een platform wordt genoemd, is het in feite een protocol of ecosysteem voor gedecentraliseerde sociale webapplicaties, dat bovenop de bestaande W3C-aanbevelingen en de Linked Data-stack werkt. [13] Met Solid hebben gebruikers elk één of meerdere persoonlijke online datastores (PODs), waarin al hun gegevens worden opgeslagen. Deze pods zijn toegankelijk via het Web en zijn onafhankelijk van toepassingen, wat betekent dat dezelfde gegevens kunnen worden hergebruikt in meerdere toepassingen. Gebruikers bepalen zelf welke toepassingen en welke mensen toegang hebben tot welke delen van hun gegevens. Dit leidt tot een grote verbetering in de bescherming van de privacy van mensen. [5]

Het Solid ecosysteem is gebouwd bovenop een aantal bestaande W3C standaarden. Het is gebaseerd op de principes van het Semantisch Web, dat het mogelijk maakt gegevens die zich overal op het Web bevinden aan elkaar te koppelen. Standaarden die door Solid worden gebruikt zijn RDF, Linked Data, SPARQL, WebID, Web Access Control, LDP en Linked Data Notifications. [5], [15], [16]

F. *Queries uitvoeren over Linked Data*

De moeilijkheid met gedecentraliseerde omgevingen die Linked Data gebruiken is het opvragen van gegevens. Vooral in situaties zoals gedecentraliseerde sociale webapplicaties wordt het verwerken van queries een uitdaging omdat het aantal onafhankelijke databronnen snel kan toenemen. Momenteel zijn er twee belangrijke benaderingen [17], [18] voor data-integratie en opvraging van data in een gedecentraliseerde omgeving.

De eerste aanpak is warehousing, ook wel materialisatie-gebaseerde aanpak genoemd. Deze methode gaat uit van één centrale database, waarin vooraf RDF-dumps van alle externe gegevensbronnen worden verzameld. Door preprocessing- en indexeringstechnieken op de geaggregeerde data toe te passen, kunnen queries eenvoudig worden beantwoord met behulp van de centrale repository. Data warehousing biedt de snelste responstijden voor queries, maar heeft een aantal grote nadelen. Eén daarvan is dat de gegevens in de centrale opslagplaats niet altijd gesynchroniseerd zijn met de bronnen.

De tweede methode is distributed query processing (DQP) of federated querying. Bij deze aanpak is geen centrale databank nodig, maar worden de gegevensbronnen rechtstreeks bevraagd. Daartoe worden de ingevoerde queries geparsed en opgesplitst in afzonderlijke subqueries, die vervolgens naar de afzonderlijke gegevensbronnen worden gestuurd, afhankelijk van het bronselectieproces. Ten slotte worden de antwoorden van de externe bronnen gecombineerd tot een geldig resultaat. Distributed query processing is op verschillende manieren voordelig. Eén voordeel is dat, in tegenstelling tot warehousing, de gegevens altijd gesynchroniseerd zijn, aangezien er slechts één kopie van is. Helaas kunnen de beschikbaarheid en betrouwbaarheid van deze aanpak niet worden gegarandeerd, omdat het systeem afhankelijk is van een groot aantal mogelijk onstabiele of inactieve data stores. [18]

G. *Bronselectie*

De meeste distributed query processing benaderingen zijn niet erg schaalbaar wanneer het netwerk van databronnen groeit. Bestaande methoden zijn vaak ontworpen om slechts een klein aantal bronnen te verwerken die een grote hoeveelheid gegevens bevatten, in plaats van duizenden afzonderlijke bronnen die elk minder resources bevatten. Het laatste geval zou echter ideaal zijn voor gedecentraliseerde sociale webapplicaties, waarbij mogelijk duizenden of miljoenen gebruikers, verspreid over de hele wereld, elk hun eigen persoonlijke datastore hebben. Maar wanneer er een zeer groot aantal gegevensbronnen is, is het onhaalbaar om elke bron te controleren op potentiële gegevens die bijdragen tot het resultaat van een query. Daarom moet een bronselectieproces worden uitgevoerd, om bronnen uit te sluiten die geen waardevolle gegevens voor de query bevatten en alleen die bronnen te behouden die kunnen bijdragen aan het antwoord op de query. Dit proces neemt een triple pattern als invoer en geeft een reeks databronnen terug die potentiële variable bindings bevatten. [18]

Twee effectieve datastructuren om bronselectie over vele bronnen mogelijk te maken zijn data summaries [16], [18] en Approximate Membership Functions (AMF's). [16], [20], [21] Ze zijn beide probabilistisch, wat betekent dat ze false negatives kunnen produceren, maar geen false positives. De kans dat false positives optreden kan worden gekozen door een false positive probabiliteit in te stellen.

Er zijn twee verschillende implementaties van AMF's, namelijk Bloom filters [22] en Golomb-coded sets. [23]

III. Implementatie

In dit onderzoek worden Approximate Membership Functions toegepast op een Solid-omgeving, om het effect ervan op de query-performantie te evalueren. De nadruk ligt op schaalbaarheid van gedecentraliseerde netwerken van databronnen, door een uitbreidend sociaal netwerk te simuleren. Met dit onderzoek hopen we iets dichter te komen bij het realiseren van grootschalige gedecentraliseerde sociale webapplicaties.

A. *Datageneratie*

Om een sociaal netwerk te simuleren waarop experimenten kunnen worden uitgevoerd, moesten veel synthetische gegevens worden gegenereerd. Hiervoor werd een gedecentraliseerde versie van de LDBC SNB Data Generator [24] gebruikt.

De gedecentraliseerde versie splitst het grote Turtle bestand, gecreëerd door de SNB data generator, zodat de data verspreid wordt over een veelvoud van bestanden, die elk de data bevatten van één specifieke entiteit in het sociale netwerk. Meer specifiek bevat elk bestand alle triples waarvan het onderwerp een specifieke URI is, die verwijst naar een entiteit in het netwerk. In dit onderzoek werden, om performantieredenen, alleen de bestanden met gegevens van personen en steden gebruikt, met een maximum van 3500 personen en de 1231 steden waarin zij wonen.

B. *Server*

Achter de schermen gebruikt de gedecentraliseerde data generator de Community Solid Server (CSS) [25] om de RDF bestanden te hosten en aan te bieden via HTTP. CSS is een open-source en modulaire implementatie van de Solid specificaties. Met behulp van de server kunnen ontwikkelaars gedecentraliseerde Solid-applicaties maken en ermee experimenteren. De gedecentraliseerde data generator tool maakt gebruik van de file-based store functionaliteit van de Solid Server om de gegenereerde data over HTTP aan te bieden. Op die manier kan een gedecentraliseerd sociaal netwerk worden gesimuleerd, aangezien de gegevens van elke afzonderlijke entiteit moeten worden benaderd via een afzonderlijke URI.

C. *Approximate Membership Functions*

Voor de constructie van de AMF's is een apart script geschreven. Het script gebruikt Bloom filters als probabilistische datastructuur. Eerst wordt het aantal bits in de bitmap van de filter en het aantal hashfuncties bepaald, gebaseerd op het aantal triples in het RDF bestand en de gewenste false positive probabiliteit. Vervolgens wordt een Bloom filter geïnitialiseerd voor elke term (subject, predicate, object). Ten slotte wordt de corresponderende term van elke triple toegevoegd aan de filter. De pseudocode hieronder toont het algoritme voor de constructie van AMF's.

```
m = calculateBitmapSize()
k = calculateNumberOfHashes()
filters = ['subject','predicate','object']
for (variable in filters):
  filters[variable] = new Bloem(m, k)
for (triple in triples):
  for (variable in filters):
    filters[variable].add(Buffer.from(triple[variable]))
for (variable in filters):
  filters[variable] = {
    type: 'http://semweb.mmlab.be/ns/membership#BloomFilter',
    filter: filters[variable].bitfield.buffer.toString('base64'),
    m: m,
    k: k
  }
```

D. *Client-side query engine*

Het belangrijkste aspect van de uitvoering van dit onderzoek was de uitbreiding van een client-side query engine om bronselectie op basis van AMF's te ondersteunen. Hiervoor hebben we de Comunica [26] query engine gebruikt. Comunica is een modulaire web-gebaseerde SPARQL query engine die het mogelijk maakt om nieuwe Linked Data query processing functionaliteiten te ontwikkelen en te testen. Het maakt federated querying over heterogene databron-interfaces out-of-the-box mogelijk.

Voor dit onderzoek werd de modulariteit van Comunica benut om bronselectie op basis van AMFs te implementeren. Een nieuwe actor werd ontwikkeld met als doel het filteren van de lijst van databronnen.

E. *Bronselectie*

Het doel van de gegenereerde Approximate Membership Functions is een bronselectieproces mogelijk te maken, waarbij veel van de irrelevante gegevensbronnen voor een gegeven query kunnen worden weggefilterd. Hierdoor blijft er een kleinere lijst met bronnen over die bevraagd moet worden, wat tot betere prestaties leidt. Hoe lager de gekozen false positive probabiliteit voor de AMF's, hoe kleiner de gefilterde lijst met bronnen. Onderstaande pseudocode toont het algoritme dat voor de bronselectie wordt gebruikt.

```
filteredSources = []
for source in originalSources:
  addSource = true
  for term in [‘subject’,’predicate’,’object’]:
    if triplePattern[term] not variable:
      if not source.bloomFilter[term].contains(triplePattern[term]):
        addSource = false
  if addSource:
    filteredSources.push(source)
```

IV. Experimentele Setup

Om de resultaten zo betrouwbaar mogelijk te maken, zijn de experimenten met veel verschillende combinaties van parameters uitgevoerd. Om te beginnen werden 15 verschillende SPARQL queries getest. Vervolgens werd het aantal databronnen in het sociale netwerk gevarieerd om een netwerk van 10, 100, 500, 1000, 2000 en 3500 bronnen te bekomen. Ten derde werden zes verschillende false positive rates getest voor de Bloom filters, namelijk 1/4096, 1/1024, 1/128, 1/64, 1/4 en 1/2. Ten slotte werd elke mogelijke combinatie van de bovengenoemde parameters driemaal herhaald. Telkens werd het gemiddelde van de metrieken van die drie iteraties berekend om het definitieve resultaat te verkrijgen.

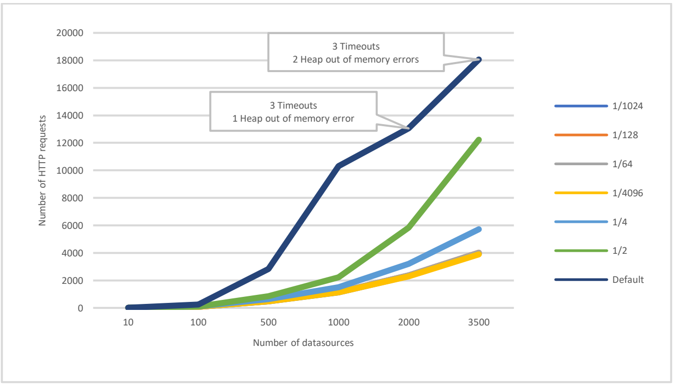
Naast het aantal resultaten dat de query opleverde, werden nog vier andere metrieken gemeten, namelijk het aantal HTTP-requests dat door de query-engine werd gedaan, de totale uitvoeringstijd van de query, het client-side geheugengebruik en de client-side CPU-belasting. Helaas leverde de CPU-belasting willekeurige waarden op, waardoor deze onbruikbaar werd in de bespreking van de experimentele resultaten.

Om het experiment reproduceerbaar te maken, werd een Bash script geschreven, dat iteratief de query engine aanroept met verschillende parameters. Op deze manier kon het hele experiment worden uitgevoerd door slechts één script uit te voeren. Een query timeout werd ingesteld op 5 minuten, in combinatie met een maximale geheugenlimiet van 4096 MB voor Nodejs. Alle experimenten werden uitgevoerd op een enkele machine met een Intel Core i7-8705G CPU aan 3,10 GHz en met 8 GB RAM.

V. Resultaten

A. HTTP-requests

De eerste en belangrijkste metriek om te evalueren is het aantal HTTP-requests dat door het queryproces wordt uitgevoerd. Figuur 1 toont het aantal HTTP-verzoeken per aantal databronnen, uitgemiddeld over alle queries (lager is beter). De legende toont de verschillende false-positive probabiliteiten van de Bloom filters, waarbij *Default* verwijst naar het gebruik van de query engine zonder AMF's. De standaard query engine wordt vanaf nu aangeduid met E, terwijl de AMF-extended query engine E' wordt genoemd.

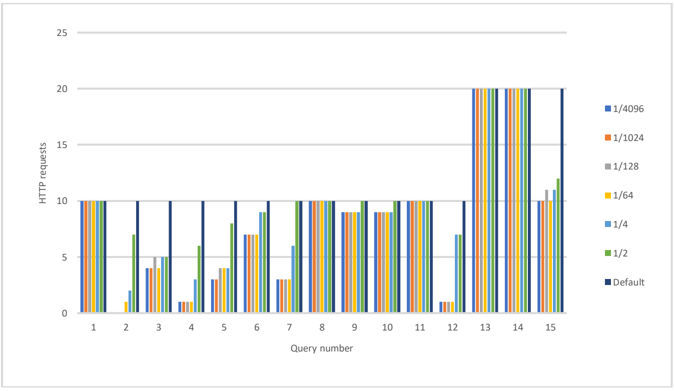


Figuur 1 – Gemiddeld aantal HTTP-requests per aantal databronnen.

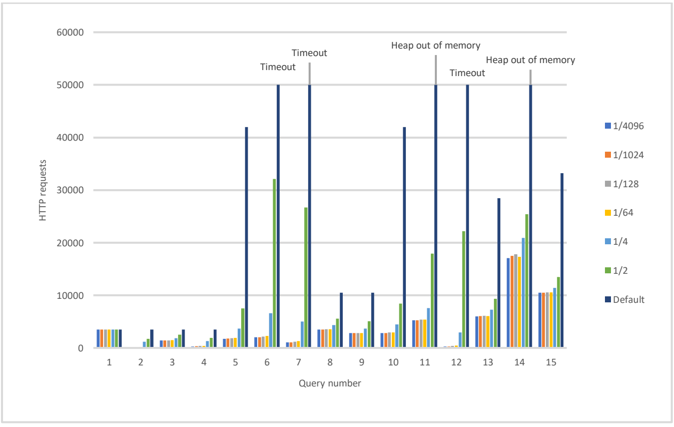
Merk op dat de grafiek niet helemaal accuraat is, omdat verschillende queries (uitgevoerd over 2000 en 3500 databronnen) resulteerden in een timeout of een *heap out of memory* error. Dit kwam alleen voor in de standaardversie van de query-engine, niet bij het gebruik van AMF's. Vanwege deze fouten zijn er geen metrieken gemeten voor die queries en dus is het aantal HTTP-requests van deze queries niet meegenomen in de gemiddelden. Daarom zijn de gemiddelden voor 2000 en 3500 databronnen met de standaardversie in werkelijkheid veel hoger en volgen ze het patroon dat is gevonden bij de lagere aantallen databronnen.

Toch valt de standaardversie van de query-engine meteen op, omdat het aantal HTTP-requests veel sneller toeneemt dan bij het gebruik van AMF's. Van 500 naar 1000 databronnen in de standaardversie resulteert in een toename van 264%, terwijl de toename bij het gebruik van AMF's bij een false positive rate $p = 1/64$ slechts 137% bedraagt. Het verschil in het gemiddelde aantal HTTP-requests tussen E en E' (bij $p = 1/64$) voor 500 databronnen is 2347 requests. Hetzelfde verschil voor 1000 databronnen is 9152 requests, wat neerkomt op een toename van 290%.

Terwijl het aantal HTTP-requests bij $p = 1/4096$, $p = 1/1024$, $p = 1/128$ en $p = 1/64$ bijna perfect op elkaar aansluiten, wordt een significante toename waargenomen bij percentages van $p = 1/4$ en $p = 1/2$. Bij 3500 databronnen voert $p = 1/2$ meer dan 3 keer zoveel HTTP-verzoeken uit in vergelijking met $p = 1/4096$. Hoge false positive rates dragen dus bij aan slechtere query-performantie.



Figuur 2 – Aantal HTTP-requests per query voor 10 databronnen.

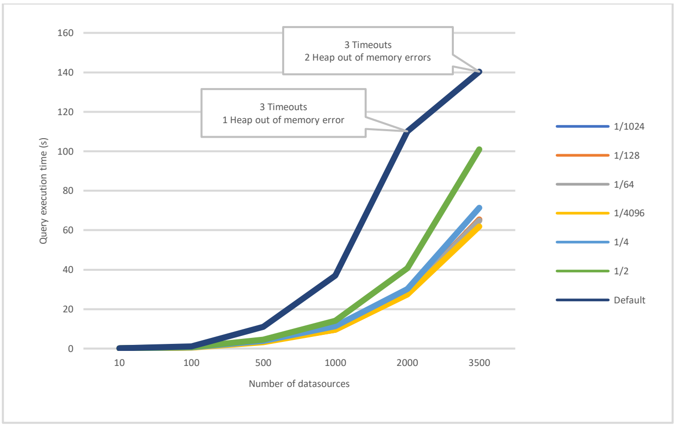


Figuur 3 – Aantal HTTP-requests per query voor 3500 databronnen.

De bovenstaande geclusterde kolomdiagrammen werden geconstrueerd om de resultaten meer in detail te onderzoeken. Deze histogrammen tonen het exacte aantal HTTP-requests voor elke uitgevoerde query (gemiddeld over 3 iteraties). Figuur 2 en Figuur 3 tonen de waarnemingen voor respectievelijk 10 en 3500 databronnen. Het is duidelijk dat een toename van het aantal databronnen leidt tot een groter verschil tussen E en E', althans voor lage false positive rates.

B. *Query uitvoeringstijd*

Naast het aantal HTTP-requests werd ook de totale responstijd van de queries gemeten. Deze metriek wordt rechtstreeks beïnvloed door de HTT-requests, aangezien een belangrijk deel van het query proces het contacteren van remote databronnen inhoudt. Figuur 4 illustreert de totale query uitvoeringstijd per aantal databronnen, uitgemiddeld over alle queries (lager is beter).



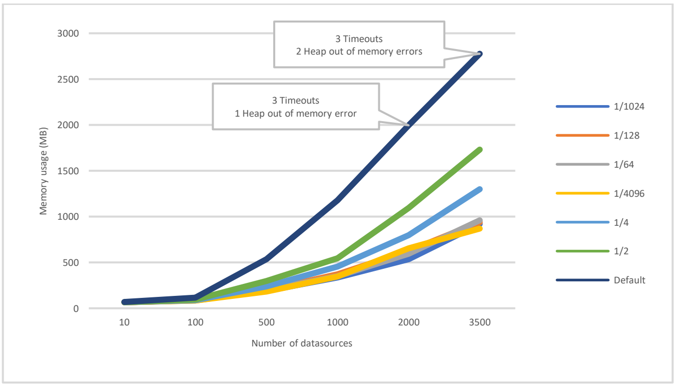
Figuur 4 – Gemiddelde query uitvoeringstijd per aantal databronnen.

Hetzelfde patroon doet zich voor als bij het aantal HTTP-verzoeken. E' met lage false positive probabiliteiten vertoont de beste responstijden, gevolgd door E' met hogere false positive rates. De gemiddelden van E liggen relatief dicht bij die van E' voor zeer kleine netwerken, maar nemen veel sneller toe als het netwerk zich uitbreidt. Van 500 naar 1000 databronnen bij gebruik van E' en $p = 1/64$ levert een toename op van 162% in de gemiddelde query responstijd. Deze toename loopt op tot 236% bij gebruik van de standaard query-engine. Het verschil tussen E en E' (bij $p = 1/64$) bij een verandering van 500 naar 1000 bronnen laat een toename zien van 273%.

False positive rates $p = 1/4$ en $p = 1/2$ laten opnieuw verminderde prestaties zien in vergelijking met lagere probabiliteiten. $p = 1/4096$ draagt bij tot iets betere resultaten dan $p = 1/1024$, $p = 1/128$ en $p = 1/64$, waarvan de gemiddelden bijna perfect op elkaar aansluiten.

De geclusterde kolomdiagrammen zijn hier weggelaten, omdat de resultaten hetzelfde zijn als bij het aantal HTTP-requests: het verschil tussen E en E' groeit ook met de uitbreiding van het netwerk van databronnen.

C. *Geheugengebruik*



Figuur 5 – Gemiddeld geheugengebruik per aantal databronnen.

De derde gemeten metriek is het geheugengebruik van de client, die de query-engine uitvoert. Net als de responstijd van de query, wordt ook het geheugengebruik sterk beïnvloed door het aantal HTTP-requests. Meer bronnen die moeten worden opgehaald betekent meer gegevens om in het geheugen te bewaren. Figuur 5 toont de evolutie van het geheugengebruik met een toenemend aantal bronnen, uitgemiddeld over alle queries (lager is beter).

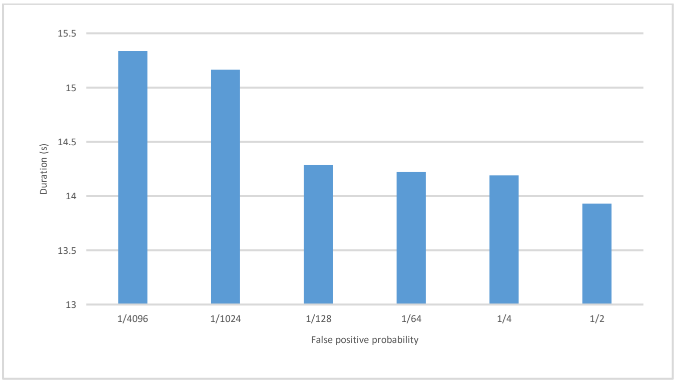
Aangezien het aantal HTTP-requests een grote invloed heeft op het geheugengebruik bij de client, kunnen we hier hetzelfde patroon waarnemen. Net als bij het gebruik van AMF's, begint het geheugengebruik van de gewone query-engine setup redelijk laag. Maar ook hier betekent het toevoegen van meer datasources een snellere toename voor E dan voor E'. Van 500 naar 1000 databronnen met de standaard engine en met $p = 1/64$ leidt tot een toename van 78% in het gemiddelde geheugengebruik bij de client. Dezelfde berekening voor de standaard engine geeft een toename van 119%. Het verschil tussen E en E' (bij $p = 1/64$) bij een verandering van 500 naar 1000 data stores neemt toe met 143%.

Lagere false positive rates laten opnieuw betere prestaties zien dan $p = 1/4$ en $p = 1/2$. Opvallend is dat de probabiliteit $p = 1/4096$ een hoger geheugengebruik veroorzaakt voor 2000 gegevensbronnen dan $p = 1/1024$, $p = 1/128$ en $p = 1/64$, maar dan weer een lager gebruik voor 3500 bronnen. Dit is waarschijnlijk te wijten aan de lagere nauwkeurigheid van de metriek voor geheugengebruik.

Ook hier zijn de geclusterde kolomdiagrammen weggelaten, omdat hetzelfde patroon kan worden waargenomen als bij de vorige twee metrieken.

D. *AMF constructie*

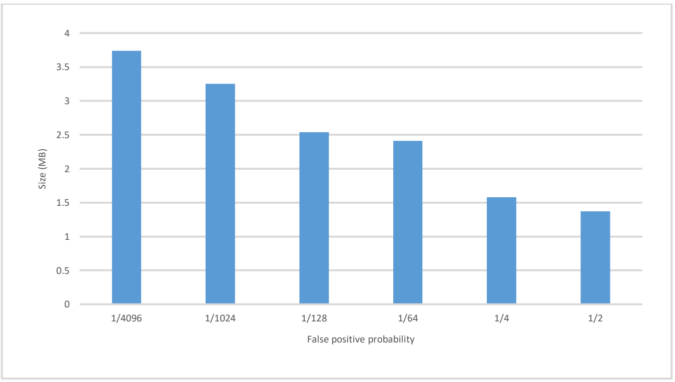
Om de prestaties van het AMF constructieproces te evalueren, zijn enkele experimenten uitgevoerd waarin de tijd die nodig is om de Bloom filters te maken is gemeten, samen met de schijfruimte die deze filters in beslag nemen. Aangezien de experimenten in de vorige paragrafen werden uitgevoerd over maximaal 3500 databronnen, die elk informatie over één persoon bevatten, worden dezelfde 3500 bronnen gebruikt in de experimenten in deze paragraaf. Er zijn echter een bijkomende 1231 databronnen toegevoegd, die elk de gegevens van een locatie bevatten. Dit brengt het totaal op 4731 gegevensbronnen voor de experimenten betreffende de AMF constructie.



Figuur 6 – Constructietijd per false positive probabiliteit voor de Bloom filters van 4731 databronnen.

Figuur 6 verduidelijkt dat een lagere false positive rate leidt tot een hogere totale constructietijd, met een maximum van 15,3 seconden voor $p = 1/4096$ (ongeveer 3 milliseconden per databron). Dit gedrag is te verwachten, want een lagere false-positive rate betekent meer precisie, en dus moet de filter meer gegevens bevatten en kost de constructie ervan dus meer tijd. Het verschil tussen de twee uitersten is relatief klein, namelijk slechts 1,4 seconden. Dit getal kan echter snel oplopen in netwerken met een schaal van honderdduizenden of zelfs miljoenen databronnen.

Als we kijken naar de schijfruimte die deze Bloom filters innemen, zien we hetzelfde patroon. Dit wordt geïllustreerd in Figuur 7. Door meer precisie toe te voegen aan de Bloom filters, moeten ze meer gegevens bevatten en meer ruimte innemen. Hoewel het verschil in constructietijd tussen de laagste en de hoogste false positive kans relatief klein is, is het verschil in omvang tussen deze twee uitersten een stuk groter. De filters met een probabiliteit van $p = 1/4096$ nemen bijna 3 keer zoveel ruimte in beslag als de filters met $p = 1/2$. In deze setting is dat verschil slechts 2,37 MB, maar in netwerken met een zeer grote schaal kan het verschil cruciaal zijn.



Figuur 7 – Schijfruimte per false positive probabiliteit voor de Bloom filters van 4731 databronnen.

In de vorige paragrafen hebben we geleerd dat lagere false positive rates het beter doen, qua prestaties. Daarentegen hebben we in deze sectie ontdekt dat Bloom filters met lagere probabiliteiten meer tijd kosten om aan te maken en meer schijfruimte nodig hebben. Daarom ligt de optimale balans tussen query-performantie, constructietijd en schijfruimte ergens tussen de hoogste en de laagste false positive rate.

Het lijkt erop dat $p = 1/64$ de aanbevolen waarde is, omdat deze veel beter presteert dan $p = 1/4$ en $p = 1/2$, terwijl er minder tijd en schijfruimte nodig is om de filters te maken dan met $p = 1/4096$, $p = 1/1024$ en $p = 1/128$ (en er qua performantie nauwelijks verschillen zijn).

VI. Discussie

Het gebruik van Approximate Membership Functions bij queries over een uitbreidend netwerk van databronnen blijkt op meerdere manieren voordelig te zijn. Het aantal HTTP-requests, als ook de totale query uitvoeringstijd en het client-side geheugengebruik worden aanzienlijk verbeterd door het gebruik van Bloom filters. We zagen dat het verschil tussen de reguliere query-engine en de query-engine uitgbreid met AMF's verwaarloosbaar is voor zeer kleine netwerken met niet meer dan 100 databronnen, maar zodra dat aantal toeneemt, neemt het verschil mee toe.

Het probleem met de huidige methode van de toepassing van Approximate Membership Functions is echter dat, zonder het gebruik van een externe server die de AMF's onderhoudt, het aantal HTTP-requests bij de client kan oplopen tot meer dan het aantal HTTP-verzoeken bij gebruik van de standaard query-engine. Dit komt doordat de Bloom filters constant up-to-date moeten zijn. Bij een input query zou de client eerst alle databronnen in het netwerk moeten contacteren om de filters te creëren. Pas daarna zou de client in staat zijn om de bronselectie uit te voeren en de query uit te voeren. Het totale aantal HTTP-requests voor elke query zou dus gelijk zijn aan de som van het aantal databronnen in het netwerk en het aantal bronnen dat bij het bronselectieproces wordt bepaald. Als gevolg daarvan zou de query-performantie slechter zijn dan met de standaard setup.

Gelukkig zijn er een paar oplossingen voor dit probleem, waarvan er hier twee zullen worden besproken. Beiden implementeren een aggregator, die gescheiden is van de client. Beide hebben ook hun eigen optimale scenario's.

De eerste oplossing is het toevoegen van een aggregator aan het netwerk, die van gegevensbron naar gegevensbron gaat en een lijst bijhoudt van alle bronnen in het netwerk. Hij kan alle AMF's periodiek vernieuwen, of hij kan een nieuwe filter genereren bij bestandswijzigingen, in welk geval de databron een notificatie kan sturen naar de aggregator. Wanneer de client een query wil uitvoeren, kan hij dit aan de aggregator laten weten, waarna de aggregator alle filters samenvoegt tot één groot bestand en dit terugstuurt naar de client. Vervolgens kan de client de bronselectie uitvoeren en de query uitvoeren. Deze methode voegt slechts één HTTP-request toe aan het volledige proces van de client. Bovendien wordt deze oplossing bijzonder interessant als de client meerdere queries over dezelfde reeks bronnen moet uitvoeren. In dat geval kan hij het gecombineerde AMF-bestand één keer downloaden van de aggregator en het hergebruiken voor verschillende queries. Deze aanpak is echter niet ideaal voor zeer grote netwerken van bronnen, omdat het bestand met de AMF's groeit met het aantal bronnen en via het netwerk naar de client moet worden gezonden. Voor $p = 1/64$ en 4731 bronnen is het versturen van een bestand met een grootte van 2,41 MB haalbaar. Uitbreiding van het netwerk tot 100.000 bronnen leidt echter tot een grootte van 51 MB, terwijl een netwerk van 1 miljoen bronnen een onhaalbare 510 MB betekent.

De tweede methode is gunstiger voor netwerken van grote schaal. In deze aanpak gaat de aggregator op dezelfde manier te werk om nieuwe bronnen te ontdekken en zijn AMF's bij te werken, maar het verschil zit in het queryproces zelf. Wanneer de client een query moet uitvoeren, zendt hij de query eerst naar de aggregator. Daarna voert de aggregator zelf het bronselectieproces uit, en zendt de lijst van geselecteerde bronnen terug naar de client. Op basis van deze lijst van bronnen kan de client de query rechtstreeks uitvoeren. Net als bij de eerste oplossing voegt deze aanpak slechts één HTTP-request toe aan het volledige queryproces van de client. Bij deze methode is de omvang van het netwerk minder belangrijk, omdat de query en de lijst van bronnen de enige gegevens zijn die met de aggregator moeten worden uitgewisseld. Deze oplossing vergt echter meer rekenkracht van de aggregator. Bovendien kan deze methode minder ideaal zijn in situaties waarin de bescherming van de privacy een zeer belangrijke rol speelt, aangezien elke query van de client over het netwerk moet worden verzonden en door de aggregator kan worden gelezen. Daarom hangt de keuze van de te implementeren oplossing vooral af van de gebruikssituatie, maar deze afweging moet in toekomstig werk verder worden onderzocht.

VII. Conclusies

In dit onderzoek hebben we geprobeerd een manier te vinden om queries over een groot aantal databronnen haalbaarder te maken. Door deze kwestie te onderzoeken zouden grootschalige gedecentraliseerde sociale webapplicaties realiteit kunnen worden. De voorgestelde methode is om de client-side query engine uit te breiden door bronselectie te implementeren op basis van Approximate Membership Functions. Door het uitvoeren van bronselectie kunnen irrelevante databronnen voor een gegeven input query worden uitgefilterd, waardoor het aantal HTTP-requests dat door de client moet worden uitgevoerd aanzienlijk wordt verminderd. Aan Approximate Membership Functions kan een false positive probabiliteit worden toegekend. Een hogere probabiliteit betekent een grotere kans dat een irrelevante bron in de geselecteerde lijst van bronnen terechtkomt. Er is echter een afweging, want AMF's met een lagere false positive rate zijn groter in omvang.

Er werden enkele experimenten opgezet om de query-performantie te meten van een query-engine uitgebreid met AMF's (E'), in vergelijking met de standaard engine (E). Het doel was te ontdekken dat het verschil in performantie tussen E' en E groter wordt naarmate het netwerk van databronnen groter wordt. Hiervoor werden drie verschillende metrieken gemeten: het aantal HTTP-requests, de totale query uitvoeringstijd, en het client-side geheugengebruik.

Voor alle drie de metrieken zagen we hetzelfde patroon. In zeer kleine netwerken zijn de metrieken van E' en E bijna gelijk. Echter, hoe meer databronnen worden toegevoegd, hoe meer het verschil in performantie tussen E' en E groeit, waarbij E' veel betere resultaten geeft dan E. Sommige queries in de reguliere setup faalden zelfs door timeouts of *heap out of memory* errors, terwijl deze queries in de AMF setup geen problemen opleverden.

Verschillende false positive rates werden ook getest, om de optimale balans te vinden tussen query-performantie, filter constructietijd en filtergrootte. We ontdekten dat een false positive probabiliteit van $p = 1/64$ bleek bij te dragen aan de beste resultaten in het algemeen.

Hoewel het gebruik van Approximate Membership Functions een aanzienlijke verbetering van de query-prestaties oplevert, moet ook rekening worden gehouden met de constructie van deze filters. In een opstelling waarbij de client zelf de AMF's moet onderhouden, is het totaal aantal HTTP-requests dat door een query wordt veroorzaakt veel hoger dan in een reguliere opstelling zonder AMF's. Hierdoor escaleren de responstijden voor queries aanzienlijk. Er zijn echter twee oplossingen voor dit probleem voorgesteld, die beide een externe aggregator impliceren.

Beide methodes voegen slechts één HTTP verzoek toe aan het volledige queryproces van de client, waardoor het volledige proces nog steeds zeer performant is in vergelijking met de reguliere query engine setup. De benaderingen verschillen echter op andere manieren en hebben elk hun eigen optimale gebruikssituaties.

Om samen te vatten kunnen we stellen dat het gebruik van Approximate Membership Functions bij queries over een groot aantal bronnen een grote verbetering van de query-prestaties oplevert, maar alleen als de middelen beschikbaar zijn om een externe aggregator te implementeren, die de AMF's beheert. Helaas, zelfs met het gebruik van AMF's, zijn de gemeten metrieken voor sommige queries over een groot aantal bronnen nog steeds behoorlijk slecht om responsieve gedecentraliseerde sociale webapplicaties mogelijk te maken. Niettemin kunnen de bevindingen in dit onderzoek ons een stap dichter bij dat doel brengen.

VIII. Toekomstig Onderzoek

Verschillende onderwerpen kunnen verder worden onderzocht om het werk van dit onderzoek voort te zetten. In de discussie hebben we twee oplossingen genoemd om het aantal HTTP-requests te verminderen, door gebruik te maken van de voorgestelde query-engine uitgebreid met bronselectie op basis van AMF's, in combinatie met een aggregator. Hoewel de optimale gebruikssituaties en de voor- en nadelen van elke aanpak kort werden besproken, moeten deze methodes verder worden onderzocht en moet er mee geëxperimenteerd worden in toekomstig onderzoek, aangezien het juiste gebruik van aggregators in een gedecentraliseerde omgeving de query-prestaties aanzienlijk kan verbeteren.

Bovendien moeten andere AMF-benaderingen worden getest, zoals gecomprimeerde Bloom filters en Golomb-coded sets. Deze datastructuren zijn efficiënter betreffende schrijfruimte dan gewone Bloom filters. Daarom zouden ze bijzonder nuttig kunnen zijn in de eerste van de twee genoemde oplossingen met betrekking tot aggregators, aangezien het grootste probleem daar is dat het verzenden van een groot AMF-bestand over het netwerk de query-performantie drastisch vermindert.

Daarnaast zijn Approximate Membership Functions niet de enige manier om bronselectie te implementeren in een client-side query engine in de context van Solid. Andere summarization-technieken moeten ook worden onderzocht en getest.

Ten slotte zouden gedecentraliseerde omgevingen op grotere schaal kunnen worden gesimuleerd, en met behulp van uitgebreidere queries kunnen worden doorzocht. We hebben nog een lange weg te gaan vooraleer we echt grootschalige gedecentraliseerde sociale webapplicaties kunnen realiseren, maar de eerste stappen zijn al gezet.

Referenties

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Sci. Am.*, 2001.

[2] N. Shadbolt, W. Hall, and T. Berners-Lee, "The Semantic Web Revisited," *IEEE Intell. Syst.*, vol. 21, no. 3, pp. 96–101, 2006, doi: 10.1109/MIS.2006.62.

[3] T. Berners-Lee, "Linked Data," Jul. 27, 2006. https://www.w3.org/DesignIssues/LinkedData.html (accessed Oct. 15, 2020).

[4] J. P. McCrae *et al.*, "The Linked Open Data Cloud," May 20, 2020. https://lod-cloud.net/ (accessed Mar. 23, 2021).

[5] A. V. Sambra *et al.*, "Solid: A Platform for Decentralized Social Applications Based on Linked Data," 2016. Accessed: Oct. 08, 2020. [Online]. Available: https://diasporafoundation.org.

[6] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," Feb. 25, 2014. https://www.w3.org/TR/rdf11-concepts/ (accessed Oct. 21, 2020).

[7] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - The story so far," *Int. J. Semant. Web Inf. Syst.*, vol. 5, no. 3, pp. 1–22, 2009, doi: 10.4018/jswis.2009081901.

[8] E. Miller, "An Introduction to the Resource Description Framework," *Bull. Am. Soc. Inf. Sci. Technol.*, vol. 25, no. 1, pp. 15–19, Jan. 2005, doi: 10.1002/bult.105.

[9] S. Harris, A. Seaborne, and E. Prud'hommeaux, "SPARQL 1.1 Query Language," Mar. 21, 2013. https://www.w3.org/TR/sparql11-query/ (accessed Oct. 24, 2020).

[10] J. Van Ossenbruggen, L. Hardman, and L. Rutledge, "Hypermedia and the Semantic Web: A Research Agenda," 2001.

[11] G. Schreiber and Y. Raimond, "RDF 1.1 Primer," Jun. 24, 2014. https://www.w3.org/TR/rdf11-primer/ (accessed Oct. 20, 2020).

[12] B. Quilitz and U. Leser, "Querying distributed RDF data sources with SPARQL," *Lect. Notes Comput. Sci.*, vol. 5021 LNCS, pp. 524–538, 2008, doi: 10.1007/978-3-540-68234-9_39.

[13] R. Buyle *et al.*, "Streamlining governmental processes by putting citizens in control of their personal data," 2019.

[14] A. Sambra, A. Guy, S. Capadisli, and N. Greco, "Building

Decentralized Applications for the Social Web," in *Proceedings of the 25th International Conference Companion on World Wide Web - WWW '16 Companion*, 2016, pp. 1033–1034, doi: 10.1145/2872518.2891060.

[15] T. Berners-Lee and R. Verborgh, "Solid: Linked Data for personal data management," Oct. 08, 2018. https://rubenverborgh.github.io/Solid-DeSemWeb-2018/ (accessed Nov. 10, 2020).

[16] R. Taelman, S. Steyskal, and S. Kirrane, "Towards Querying in Decentralized Environments with Privacy-Preserving Aggregation," 2020.

[17] P. Haase, T. Mathäß, and M. Ziller, "An evaluation of approaches to federated query processing over linked data," *ACM Int. Conf. Proceeding Ser.*, no. January, 2010, doi: 10.1145/1839707.1839713.

[18] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over Linked Data," *World Wide Web*, vol. 14, no. 5, pp. 495–544, 2011, doi: 10.1007/s11280-010-0107-z.

[19] H. Stuckenschmidt, R. Vdovjak, G. J. Houben, and J. Broekstra, "Index structures and algorithms for querying distributed RDF repositories," *Thirteen. Int. World Wide Web Conf. Proceedings, WWW2004*, pp. 631–639, 2004, doi: 10.1145/988672.988758.

[20] M. Vander Sande, R. Verborgh, J. Van Herwegen, E. Mannens, and R. Van de Walle, "Opportunistic linked data querying through approximate membership metadata," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9366, no. Iswc, pp. 92–110, 2015, doi: 10.1007/978-3-319-25007-6_6.

[21] R. Taelman, J. van Herwegen, M. Vander Sande, and R. Verborgh, "Optimizing approximate membership metadata in triple pattern fragments for clients and servers," *CEUR Workshop Proc.*, vol. 2757, pp. 1–16, 2020.

[22] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970, Accessed: Mar. 22, 2021. [Online]. Available: https://doi.org/10.1145/362686.362692.

[23] F. Putze, P. Sanders, and J. Singler, "Cache-, Hash-, and space-efficient bloom filters," *ACM J. Exp. Algorithmics*, vol. 14, Sep. 2008, doi: 10.1145/1498698.1594230.

[24] R. Taelman, "rubensworks/ldbc-snb-decentralized: A tool to create a decentralized version of the LDBC SNB social network dataset, and serve it over HTTP.," 2021. https://github.com/rubensworks/ldbc-snb-decentralized (accessed Mar. 19, 2021).

[25] R. Verborgh, J. Van Herwegen, and R. Taelman, "solid/community-server: Community Solid Server: an open and modular implementation of the Solid specifications." https://github.com/solid/community-server (accessed Mar. 20, 2021).

[26] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, "Comunica: a Modular SPARQL Query Engine for the Web," *Proc. 17th Int. Semant. Web Conf.*, vol. 11137, pp. 239–255, 2018.

Table of Contents

List of Figures

List of Tables

Listings

List of Abbreviations

ACL	Access Control List
AMF	Approximate Membership Function
API	Application Programming Interface
CPU	Central Processing Unit
CSS	Community Solid Server
DC	Dublin Core
DL	Direct Lookup
DQP	Distributed Query Processing
FOAF	Friend of a Friend
GCS	Golomb-Coded Sets
GDPR	General Data Protection Regulation
HTTP	Hypertext Transfer Protocol
II	Inverted URI Indexing
IRI	Internationalized Resource Identifier
LDN	Linked Data Notifications
LDP	Linked Data Platform
LOD	Linking Open Data
MB	Megabytes
MDH	Multidimensional Histograms
OIDC	OpenID Connect
OWL	Web Ontology Language
POD	Personal Online Data store
QT	QTree
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SLI	Schema-level Indexing
Solid	Social Linked Data
SPARQL	SPARQL Protocol and RDF Query Language
TLS	Transport Layer Security
TPF	Triple Pattern Fragments
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WAC	Web Access Control
WoD	Web of Data
XML	eXtensible Markup Language

1. Introduction

Data on the Web is now more prominent than ever. Using data, we can communicate with each other, find the nearest grocery store, order pizza, and so much more. To enable not only humans, but also machines to actually understand information on the Web, the Semantic Web [1], [2] was introduced. This extension of the Web makes data machine-readable by giving meaning or *semantics* to it.

In this context, the idea of Linked Data [3] was developed, which encourages people to publish and interlink as much data as possible, following predefined guidelines. The result is a large Web of Data (WoD), which is expanding exponentially. The Linking Open Data (LOD) cloud [4], an initiative that visualizes this WoD, currently contains 1255 datasets interconnected by 16174 links.

This concept of Linked Data opens up a new world of possibilities. One of these possibilities, which has already been brought into reality, is the Social Linked Data (Solid) project. [5] Solid is an ecosystem that defies the traditional method of centralized data storage by providing users with their own personal online data store (POD). As it is based on the principles of Linked Data, information stored in the pods can be linked, creating a network of personal data stores. This enables the realization of decentralized social Web applications, which provide some major advantages in comparison to their centralized counterpart. For one, giving users control over their own data significantly improves their privacy. Secondly, storing a user's data at a single location avoids duplication of data and lets users reuse information across multiple decentralized social Web applications.

However, in the case of large-scale decentralized Web applications, data might be spread across thousands or millions of remote datasources. In contrast to centralized social networks, where all data is fetched from a single database, retrieving data in a large-scale decentralized environment is not that straightforward. To this end, in this research we will try to find a way to make efficient data retrieval in decentralized networks of personal data stores more attainable.

We will rely on the implementation of a source selection process in a client-side query engine to achieve this goal. Source selection reduces the number of datasources to be contacted for a given query, by ruling out irrelevant datasources for that query. More specifically, we will use Approximate Membership Functions, which are small probabilistic data structures, to enable source selection.

Related concepts and research will first be introduced in the literature study. In section 3, the problem statement and motivation for this research will be clarified, followed by an explanation of the used methods and technologies in section 4. The results of the performed experiments will be analyzed in section 5 and will be further discussed in section 6. Finally, we will draw some conclusions in section 7 and discuss possible future work in section 8.

2. Literature Study

In this research, we try to find an efficient way to collect decentralized data from many distributed data sources and display it to end users. In order to do this, various concepts and technologies must be studied, all associated with the Semantic Web.

First, the Semantic Web will be explained, followed by the concept of Linked Data. Then, the Resource Description Framework (RDF) and its query language, SPARQL, will be investigated. To conclude, we will study the Social Linked Data (Solid) project and look at existing solutions for querying over Linked Data.

2.1 The Semantic Web

The World Wide Web is a vast and ever-growing ocean of documents and links. Anyone can upload content to this enormous web, which can be represented as a large directed graph. [6] Unfortunately, most of the content on the Web is only human-readable. Computers can merely display the information but have no notion of the actual meaning or semantics of it. That is why the Semantic Web [1], [2] was invented.

The Semantic Web is an idea which Tim Berners-Lee, who also invented the World Wide Web, came up with. It is an extension of the current Web, in which information can be processed, understood, and manipulated by computers. The idea behind it is to give meaning to data by applying a well-defined structure to it. RDF [7], a language that represents information in the Web, expresses this meaning. It does this by uniquely identifying any object and interlinking these objects to form webs of information. [1]

When the technologies supporting the Semantic Web have evolved enough, a broad variety of possibilities becomes within reach. One of these possibilities is the concept of software agents [1]. These are pieces of software that aggregate information from various sources, process the data and then communicate the output with other agents. The use of these intelligent agents could alter the way we deal with the Web drastically. For instance, using a software agent compatible with the Semantic Web, one could automatically set up a doctor's appointment by letting the agent browse through the Web and gather information about opening hours, distances, ratings, and so on, while taking the personal calendar into account. However, before such ideas can become reality, a lot must change about the Web. In the following paragraphs, we will examine some related concepts more in detail.

2.2 Linked Data

The idea of connecting all related data to form a Web of Data is referred to as Linked Data by Tim Berners-Lee. [3] It is a set of rules for publishing and linking information on the Web. Initially, the Web consisted of linked documents, without machines being able to comprehend the actual meaning of the information. Now, the Web is evolving into a global space in which both documents and data are connected. [8] This upgraded version of the Web has many advantages and applications. Linked Data search engines, for example, have the ability to aggregate data from many data sources by following links between them. The accumulated data can then be queried.

Tim Berners-Lee formulated four rules [3], known as the Linked Data principles, for publishing new data on the Web, with the purpose of contributing to the Web of Data:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs, so that they can discover more things.

As the first rule suggests, everything that wants to be a part of the Web of Data, must be defined by a Uniform Resource Identifier (URI). This is because a URI is a unique identification and it can easily be parsed by a computer. [9] By using Hypertext Transfer Protocol (HTTP) URIs, as the second rule states, a name lookup can be performed over HTTP. The third rule says that the standard Semantic Web technologies such as RDF and SPARQL must be used when transferring and presenting data. These concepts have been designed to support the evolution of the Semantic Web and will be discussed in paragraphs 2.3 and 2.4, respectively. The last rule emphasizes the importance of linking the data to other data already in the Web, with the purpose of expanding the Web of Data.

2.2.1 The Linking Open Data Community Project

The goal of linked data is to make all data on the Web accessible to everyone, including computers, and to establish links between all related data. To stimulate this idea, the Linking Open Data community project [8], [10] was founded under the auspices of W3C. The aim of this project is to publish existing data sets converted into RDF and creating links between these various data sources. This is an open project, so anyone can add a data set to the Web of Data conforming to the Linked Data conventions and link it to the existing sets.

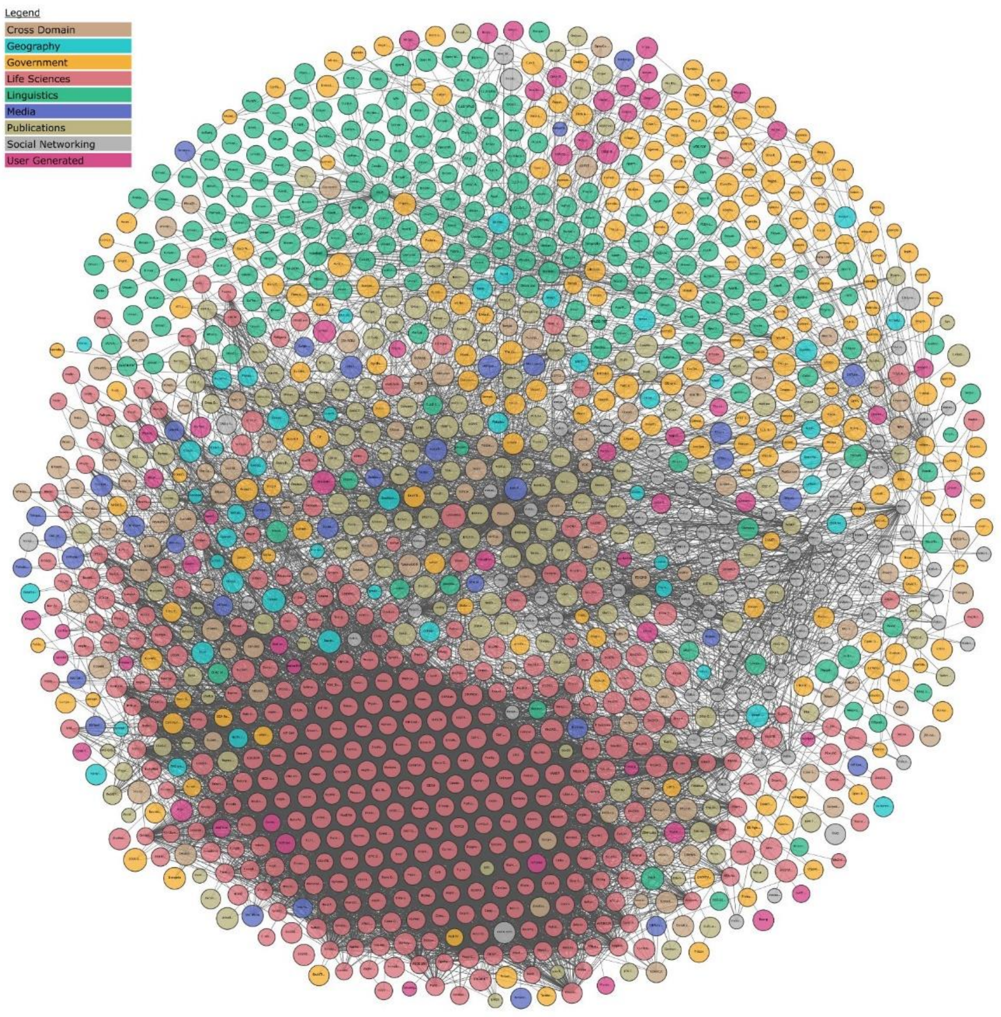


Figure 1 – The Linking Open Data cloud in 2021. Source: [4]

Figure 1 illustrates the current Linking Open Data cloud. It is a graph in which the nodes are data sets published as Linked Data, and the edges depict the relations between items in the data sets. [8] This web is continuously growing, with currently 1255 data sets and 16174 links in it, in comparison to only 203 data sets in 2010. [4]

2.2.2 Linked Data Platform

Linked Data Platform (LDP) [5], [11], specified by W3C, defines a set of guidelines for the use of HTTP to access, create, update and delete Web resources from servers that rely on Linked Data to expose their resources. The protocol provides clients and servers to read and write resources using the representational state transfer (REST) architectural style and HTTP Methods such as GET and POST. Linked Data Platform Resources (LDPRs) may or may not be RDF-based, with HTML files and images being examples of LDP Non-RDF Sources. Collections of LDPRs are called Linked

Data Platform Containers (LDPCs). These LDPCs are in fact also LDPRs, which makes creating a hierarchy of nested LDPCs and LDPRs possible. In practice, an LDPC is a collection of links to LDPRs, allowing applications to follow paths through the resources.

2.3 The Resource Description Framework

We have access to all sorts of information through the World Wide Web. The way to discover such information is via metadata, or structured data about data. However, this metadata is used everywhere, so certain conventions about syntax, structure and semantics are necessary to achieve a global overview. The Resource Description Framework, specified by the World Wide Web Consortium (W3C), defines these conventions. RDF allows metadata to be encoded, exchanged, and reused. [12] This will later be made clear.

RDF presents a data model optimized for linking data that defines resources. This model is graph-based, which means it can be visualized by a directed labeled graph. This makes RDF suitable for representing social network data. [8], [13] A resource is an object that can be identified by a URI[1]. [14] When Hypertext Transfer Protocol (HTTP) URIs are used, these resources can be looked up on the Web. [3]

The encoding of data mentioned above is done as triples. These triples consist of a subject, a predicate (also called a property type) and an object. The subject (a resource) and predicate of a triple are represented by URIs, whereas the object may be a URI or a string. [8] A predicate and an object together form a property of a subject. [12] The relation between a subject and an object is determined by the predicate. A Web of Data is established when for instance a subject from one dataset is linked to an object from another dataset. In this case, the RDF triple becomes an RDF link. Both the subject and the object must then be URI references. [8] Below, an example is shown to illustrate the application of the RDF data model.

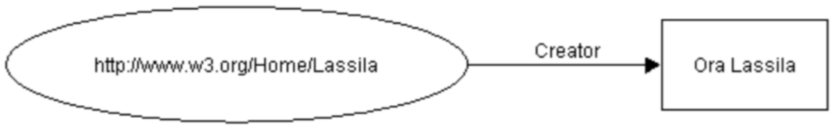


Figure 2 - RDF triple represented as an RDF graph. Source: [15]

In Figure 2, a subject identified by the URI *http://www.w3.org/Home/Lassila* is related to an object identified by the string literal *Ora Lassila*. The relation is that the object is the creator of

[1] Currently, the term Internationalized Resource Identifier (IRI) is used instead of URI. IRIs are a generalized version of URIs, that accept more Unicode characters. [7]

the subject, as the predicate suggests. Here, the predicate is no HTTP URI for the purpose of keeping things simple in this example. The object can, however, also be a resource. That way, the object can become a subject of a new triple, which means that it can have its own property types and corresponding values. This makes it possible for the RDF graph to expand.

It is important for resources to have unique identifiers, as it allows them to be unambiguously defined. This in turn implies that properties may be reused. [12] In the previous example, *Ora Lassila* can be the creator of many resources. By uniquely identifying this person with an URI, the object becomes a resource that can be used throughout the RDF graph. This prevents unnecessary duplication of objects.

For data to be exchanged and processed, a general syntax or data serialization format is required. [12] Using such a format, an RDF graph can have a textual representation. There are several RDF syntaxes, including RDF/XML, N-Triples, Turtle and JSON-LD. [16] Originally, RDF applied the eXtensible Markup Language (XML) as its syntax. [12] It was designed to be used throughout the Internet [17] and it allows for RDF to represent its semantics consistently and unambiguously. The XML syntax can convert instances of the RDF data model into machine-readable files so that they can be exchanged between applications. [12]

XML provides a system called XML namespaces, which supports the unambiguous representation described earlier. XML namespaces help property types have a unique identification. This is convenient because one property type having multiple meanings would counteract the unambiguity of the semantics. [12] RDF vocabularies (or ontologies [18]) are what XML namespaces use to assign these unique identifications. Vocabularies are sets of classes and properties and are represented using RDF. [8] Various resource description communities define these vocabularies to give their own meaning to property types. [12] Anyone can publish RDF vocabularies on the Web [18] and relations between vocabularies can be established by connecting them through RDF links. [8] The Web Ontology Language (OWL) [19] and the Resource Description Framework Schema (RDFS) [20] form a basis for constructing new vocabularies. They already specify numerous classes and properties to describe resources and their relations, especially the rich and widely adopted OWL.

```
<?xml:namespace version="1.0" encoding="utf-8"?>
<rdf:RDF
        xmlns:dc=”http://purl.org/dc/terms/”
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns">
    <rdf:Description rdf:about=”http://www.w3.org/Home/Lassila”>
        <dc:creator>Ora Lassila</dc:creator>
    </rdf:Description
</rdf:RDF>
```

Listing 1 - RDF triple in RDF/XML syntax

The code in Listing 1 is a textual representation of the RDF graph in Figure 2, using the RDF/XML syntax. Note that the predicate *creator* is now a property type specified by the Dublin Core schema (DC). Some namespace prefixes have been defined to make the code more readable.

The XML syntax, however, presents some unwanted overhead [21], and some new formats have been developed since RDF was invented. N-Triples [16] is the simplest format, in which each individual line contains an RDF triple. URIs are in between brackets (<>) and each line is ended with a period.

```
<http://www.w3.org/Home/Lassila> <http://purl.org/dc/terms/creator>
<http://www.w3.org/staffId/85740> .
```

Listing 2 - RDF triple in N-Triples syntax

In Listing 2, the previously introduced RDF triple is represented in N-Triples syntax. The string literal *Ora Lassila* is now replaced by a URI, uniquely identifying that person.

The Turtle syntax [22] is an extended version of the N-Triples format. It includes features like namespace prefixes, predicate lists, and object lists.

```
@prefix DC: <http://purl.org/dc/terms/> .

<http://www.w3.org/Home/Lassila> DC:creator
<http://www.w3.org/staffId/85740> .
```

Listing 3 - RDF triple in Turtle syntax

Listing 3 represents the triple in Turtle syntax. It uses the prefix *DC* as reference to the Dublin Core vocabulary, which makes the code more readable. Prefixes can be reused among multiple triples, so that typing the entire URI every time becomes unnecessary.

Another extension of N-Triples is the N-Quads format. As the name suggests, a fourth element is added at the end of a line, being the graph IRI of the corresponding triple. This allows for exchanging RDF datasets.

2.4 SPARQL

Data encoded in the RDF format does not have much use if it cannot be queried in some way. For that matter, the SPARQL Protocol and RDF Query Language (SPARQL) [13], [23] was developed by W3C in 2008. SPARQL is a query language for RDF encoded data and can be used for querying multiple graphs. In addition to the possibility of executing SPARQL queries over RDF dumps, content providers can set up a SPARQL endpoint. A big advantage of these endpoints is that the data does not necessarily need to be stored in RDF, as it can be created at run-time from other data sources.

A SPARQL query consists of a basic graph pattern, which is a series of triple patterns. These are much like RDF triples, but they can contain variables. The subject, the object or the predicate of the triple pattern may be replaced by a variable. When RDF terms (RDF literals, IRIs, or blank nodes) from a subgraph can be replaced by variables, resulting in a graph equal to that subgraph, then the basic graph pattern matches the subgraph. A successful SPARQL query results in a sequence of solutions, specified by the variables present in the `SELECT` clause. This sequence can be represented as a table, with one solution per row. The basic graph pattern is defined in the `WHERE` clause. Note that for every solution, all the variables mentioned in the `WHERE` clause must bind to the RDF terms of the solution.

```
@prefix DC: <http://purl.org/dc/terms/>

SELECT ?page ?name
WHERE
{
  ?page DC:creator ?name .
}
```

Listing 4 - A SPARQL query

page	name
<http://www.w3.org/Home/Lassila>	"Ora Lassila"

Table 1 - Result of a SPARQL query

Listing 4 shows an example of a simple SPARQL query. It extracts data from the previously seen RDF triple, only that the object is the string literal *Ora Lassila* again, instead of a URI. The `SELECT` clause determines that the solutions should include the RDF terms bound to the variables *page* and *name*. In the `WHERE` clause, it is stated that we want to query RDF triples that contain any subject (bound to the *page* variable), related to any object (bound to the *name* variable) via the *DC:creator* predicate.

The result of this query is illustrated in Table 1. As there is only one RDF triple in the data source, there cannot be more than one solution. The triple with subject *http://www.w3.org/Home/Lassila* and object *Ora Lassila* matches with the only triple pattern in the query, and so it matches with the basic graph pattern. SPARQL queries can be made much more extensive to allow for complex result sets, but this is outside the scope of this research.

Query results can take on multiple forms, specified by the SPARQL query forms [13]. `SELECT` is the simplest form and returns variable bindings. If an RDF graph is desired instead of a table, the `CONSTRUCT` clause can be applied. This form needs a triple template, in which the requested variables are substituted, to build a result graph. The `ASK` query form returns *true* when the query pattern matches the data and returns *false* when it does not. Finally, the `DESCRIBE` form returns an RDF graph describing the matched resources.

2.5 Solid

Deprivation of privacy is an important problem nowadays, especially when it comes to personal data gathered by all sorts of companies and governments. Since the introduction of the European General Data Protection Regulation (GDPR), people have become more aware of businesses collecting personal information. Unfortunately, while companies outside of Europe also must be compliant to the GDPR for their European clients, many of them do not fully conform to the regulations. [24]

Centralized social Web applications such as Facebook and Twitter pose additional problems, for users as well as for application developers. Companies behind social networking websites each have their own databases in which they manage their users' data. This means that users cannot let similar applications reuse the same data, so they must duplicate their data across several of these social services. Social Web application developers can build upon existing platforms for reading and writing data and for handling access to it. However, such platforms govern their own Application Programming Interfaces (APIs) which developers must conform to, eliminating the freedom of creating custom solutions. [5]

To tackle these issues, the Social Linked Data (Solid) project [5] was developed, which puts users in control of their own data. Although referred to as a platform, Solid is in fact a protocol or ecosystem for decentralized social Web applications, working on top of existing W3C recommendations and the Linked Data stack. [24] With Solid, users each have one or more *personal online data stores* (PODs), in which all of their data is stored. These pods can be accessed through the Web and are independent from applications, which means that the same data can be reused throughout multiple applications. Users decide which applications and which people have access to which parts of their data. This results in a major improvement in protecting people's privacy. [5]

Not only users of Solid applications, but also developers benefit from this decentralized approach. They do not have to worry about managing and storing user data, as this is done on the users' pods. Secondly, developers can adopt data created by other applications into their own, thereby enhancing the user experience. [25]

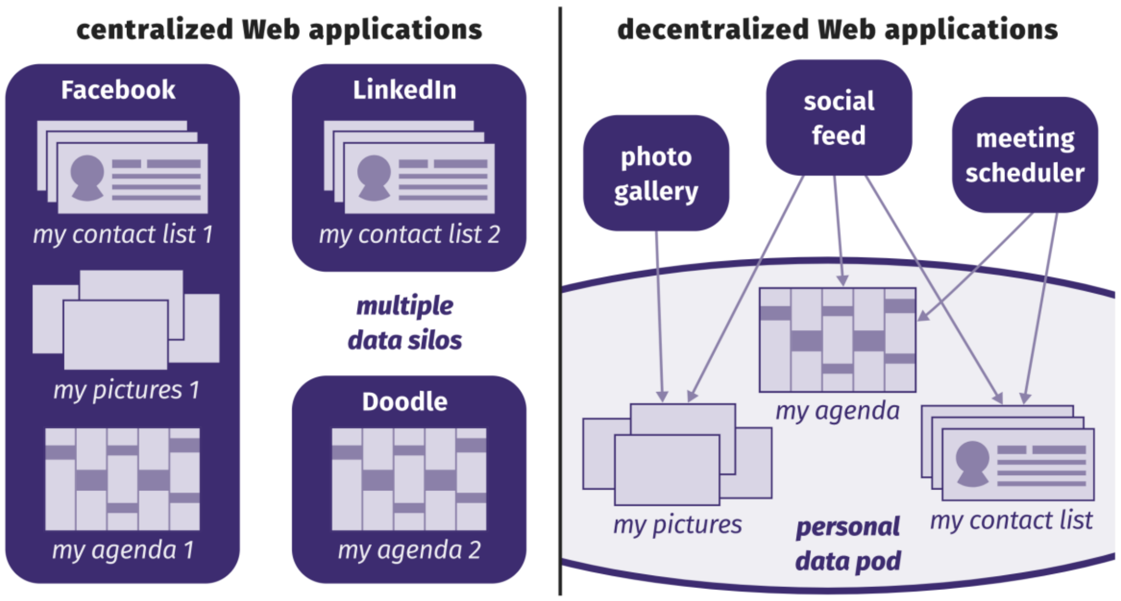


Figure 3 - Centralized vs. decentralized Web applications. Source: [26]

The difference between centralized and decentralized Web applications is depicted in Figure 3. In centralized solutions, application and data are combined into one service. This limits interaction between applications. In contrast, decentralized applications can use all data available in its users' personal data stores when granted access. Pod owners can choose to expose specific parts of their data to any application or person. [24]

When working in a decentralized way, social Web applications must combine data from multiple pods to provide their services. Posts and comments, for instance, are stored on the data pod of the user who created them. When a feed of a user's friends' activities must be visualized, all the pods of that user's friends are queried, in order to aggregate the requested data. By directly communicating with the data source, problems concerning synchronization are eliminated, as the data will always be up to date. [24]

Data pods can either be hosted on a person's own server or on public servers provided by third party services, and users are free to switch between providers at any given time. Solid applications can communicate with any pod, regardless of where it is physically located or who hosts it. [5] This opens up a new market for pod providers, in which competition is based on service quality, as is illustrated in Figure 4. Providers can distinguish themselves from others by excelling in various domains, such as reliability, storage space, security and transfer speeds. [5] As such, it is possible for users to have different pods for different purposes. [24] Currently, there are three available personal data store providers, *solidcommunity.net*, hosted by *Digital Ocean*, *inrupt.net*, hosted by *Amazon*, and *solidweb.org*, hosted by *HostEurope*. Each of these provide their services for free. [27]

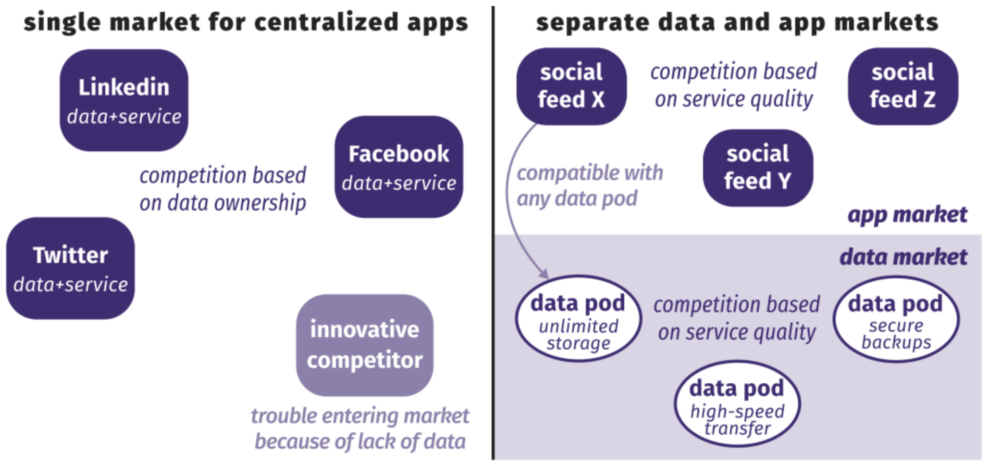

single market for centralized apps
Linkedin data+service
Facebook data+service
competition based on data ownership
Twitter data+service
innovative competitor
trouble entering market because of lack of data
separate data and app markets
social feed X
competition based on service quality
social feed Z
compatible with any data pod
social feed Y
app market
data market
data pod unlimited storage
competition based on service quality
data pod secure backups
data pod high-speed transfer

Figure 4 - Market comparison for centralized and decentralized Web applications. Source: [26]

2.5.1 The Solid Technology Stack

Everyone that wants to use Solid apps, has to take care of his or her own data storage. However, a typical aspect of social network applications is that they require lots of links and relations between users' data, in order to make it 'social'. In centralized Web applications, these links can easily be established, as all data resides in the same database. With decentralized applications, and thus, possibly thousands of different data sources, another method must be applied to obtain the same result. [5], [26]

With that in mind, the Solid ecosystem was built on top of some existing W3C standards. It relies on the principles of the Semantic Web, which allows for interlinking data that resides anywhere in the Web. Standards used by Solid include RDF, Linked Data, SPARQL, WebID, Web Access Control, LDP and Linked Data Notifications. [5], [26], [28] The purpose of these technologies within the Solid protocol will be clarified in the following sections.

2.5.1.1 Linked Data and RDF

Using the RDF data model and the Linked Data principles, any data in one pod can link to any data in another pod. [26] In addition, data is given a globally recognized meaning, with the help of ontologies. [12] Every piece of data in Solid is uniquely identified by a URI. These can either be structured or unstructured resources. Structured data is represented using the RDF language and can be serialized using N-Triples, Turtle, JSON-LD or other syntaxes. Unstructured data includes resources unserializable by RDF syntaxes, such as images and videos. By identifying these resources with URIs, RDF triples in one data source can reference triples in other data sources. That way, a Web of Data is formed across all available personal data stores. [5]

2.5.1.2 Linked Data Platform

Establishing links between related data is only a part of the solution. Solid-based applications must also be able to access and interact with this data. The previously discussed Linked Data Platform is one way to do this. It manages data in a RESTful way, by communicating with the APIs of the data sources. Creating a new resource in an LDP Container can be done by executing an HTTP POST method towards the URL of that container. Updating items happens with HTTP PUT or HTTP PATCH and deleting them is done using HTTP DELETE. HTTP GET is the method of choice for accessing all resources in a container or for obtaining particular items. Resources can also be found by following links. [5], [11], [28]

2.5.1.3 Linked Data Notifications

Solid pods do not only communicate with applications, they can also communicate with each other, using Linked Data Notifications (LDN). [28], [29] LDN is a subset of the Linked Data Platform. It makes use of HTTP methods to send and receive notifications. The communication process is illustrated in Figure 5 and happens as follows:

- When someone or some automated process wants to send a notification, it provokes a *sender* (a certain data pod) to instantiate a notification body.
- The *target* is the resource which the notification is meant for. This can be any resource, for example a user profile. Every target has an *inbox*, which corresponds to a container in LDP terms, and contains all the notifications meant for the target. The inbox URL can be discovered by issuing an HTTP GET or HEAD request to the URL of the target. The response graph then includes a triple that has *ldp:inbox* as its predicate. The subject of this triple is the inbox URL.
- The notification, serialized using a supported RDF syntax, is then sent to the inbox through an HTTP POST.
- The *receiver* manages GET requests to the inbox or to individual notifications, as well as POST requests to create new notifications.
- Finally, notifications in the inbox of the target can be requested by *consumers*. If a consumer sends an HTTP GET to the inbox, the receiver responds with the URLs of all available notifications in the inbox. If the request is addressed to a specific notification, then that notification is returned in a supported syntax. Inbox URL discovery is analogous to the sender's method.

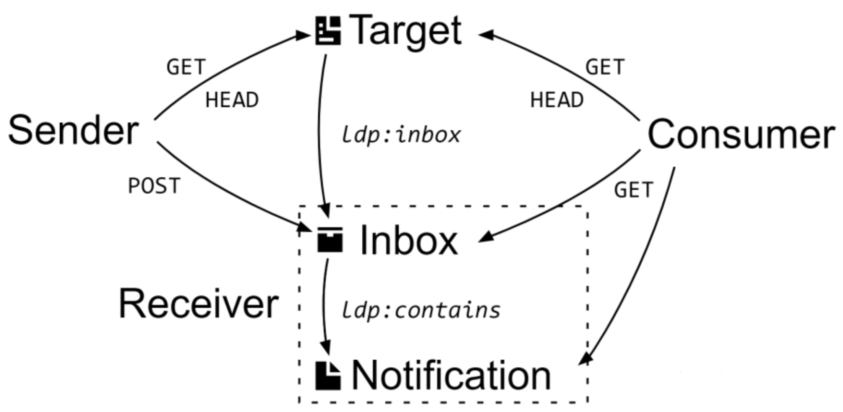


Figure 5 - Linked Data Notifications diagram. Source: [29]

2.5.1.4 SPARQL

Some Solid applications require data retrieval operations that are more complex than the methods that LDP supports. SPARQL allows developers to implement these more advanced tasks. Applications can build SPARQL queries and let Solid servers that support this query language handle them. These queries are classified into two categories, being local and link-following queries. Local queries only collect data that is present on the pod that executes the query, while link-following queries retrieve data from multiple different data sources, by following RDF links across pods. This results in a considerable reduction of HTTP requests and communication between client and server, in comparison to LDP. Another strength of this approach is that the actual distribution of the data stores and their data must not be known in advance to execute such queries. As such, it is not necessary to mention external pods in a link-following query. [5]

Instead, some preprocessing must be done to determine the servers that contain the right data to answer the query. This process is called source selection. The server that receives the input query, analyses it and breaks it down into subqueries, which are then delivered to the correct pods through HTTP, according to the source selection process. Servers that support SPARQL queries expose a SPARQL endpoint, to which queries are sent. By issuing an HTTP GET or HEAD to a pod's URI, the URI of the endpoint can be acquired from the response's headers. [5], [28]

When the data sources are known, for example through source selection, then the SPARQL `SERVICE` extension [30] can be used to perform a federated query. By applying this extension, a query processor can address certain parts of the query to specific endpoints. After query execution, the processor aggregates the results into one RDF graph.

Continuing with the previous example concerning the person named Ora Lassila, imagine that person has a personal pod with the URI *http://oralassila.com/*, which contains a file named *friends.ttl*. This file holds a list of RDF triples in Turtle syntax representing the friendships of Ora Lassila:

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

<http://oralassila.com/me> foaf:knows <http://people.org/person4> .
<http://oralassila.com/me> foaf:knows <http://people.org/person10> .
<http://oralassila.com/me> foaf:knows <http://people.org/person23> .
```

Listing 5 - RDF triples representing friendships

Notice that the objects of the triples refer to a remote server, outside of Ora Lassila's pod. This data source with the URI *http://people.org/* exposes a SPARQL endpoint to which queries can be addressed, and stores triples mapping URIs of people to their names:

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

:person3  foaf:name  "John" .
:person4  foaf:name  "Tim Berners-Lee" .
...
:person10 foaf:name  "Jane" .
...
:person23 foaf:name  "Lisa" .
:person24 foaf:name  "Craig" .
```

Listing 6 – RDF triples mapping URIs to names

If Ora Lassila wants to retrieve the names of his friends, he can do so by performing a SPARQL federated query, in which the external data source is mentioned next to the `SERVICE` keyword:

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

SELECT ?name
FROM <http://oralassila.com/friends.rdf>
WHERE
{
  <http://oralassila.com/me> foaf:knows ?person .
  SERVICE <http://people.org/sparql> {
    ?person foaf:name ?name . }
}
```

Listing 7 - A SPARQL federated query

This query first searches in the local *friends.ttl* file for occurrences of triples of which the subject is *http://oralassila.com/me* and the property is *foaf:knows*. For the second part of the query, the query processor contacts the external SPARQL endpoint, to obtain bindings for the *name* variable. The query eventually returns three bindings:

name
“Tim Berners-Lee”
“Jane”
“Lisa”

Table 2 - Result of a SPARQL federated query

To demonstrate the scalability and performance of link-following SPARQL query execution, an experiment [5] using synthetic RDF data was performed. In this experiment, the time to complete certain SPARQL queries was measured, with an increasing number of Solid pods. The observation resulted in a sublinear curve, as the results showed that it took only slightly more time to collect data from 128 pods, than it took for 2 pods. This is due to the decomposition of the input query into subqueries, which parallelizes computations across data stores. This experiment confirms that link-following SPARQL is especially fit for querying large-scale data source distributions.

To obtain data from pods using queries, these pods have to store the data in some way. Luckily, there are multiple ways to do this. RDF databases or *triple stores* [2] are most suitable for storing large amounts of RDF triples and querying them using SPARQL. With this option, non-RDF resources must be stored somewhere else, but the triple store can still contain their metadata. The file system is another frequently used implementation. This approach allows for both RDF and non-RDF resources to be stored. In this case, all resources are saved as files. In addition, this implementation facilitates the use of LDP. Pods which implement a file system may also be upgraded to tolerate SPARQL queries. [5]

2.5.1.5 WebID

User authentication in Solid, just like data storage, works in a decentralized manner. The user’s identity is not authenticated against the application, but against the data pod. Solid relies on the WebID technology for this, which contributes to a global identification system, designed for use in a decentralized social Web. [5] A WebID is a personal identifier in the form of an HTTP URI. It holds a reference to an *agent*, which can be a person, a device, an organization, et cetera. An agent’s *Profile Document* is a web page that describes the agent and is available in an RDF format. The Profile Document’s URI can either be acquired by leaving out the WebID’s *fragment identifier*, if one is available, or by sending an HTTP request to the WebID. In the latter case, the WebID Profile can be obtained from the response’s headers. [31] Figure 6 illustrates the relations between an agent, his WebID and his Profile Document.

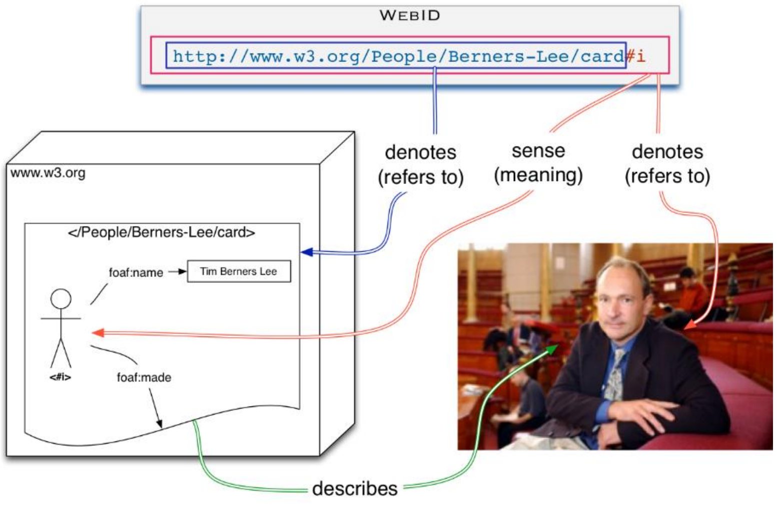


Figure 6 - WebID and Profile Document. Source: [31]

WebID Profiles can be publicly or privately interlinked using vocabularies such as the Friend of a Friend (FOAF) ontology, such that a web of trust can be established. That way, services can decide whether an agent is authorized to access certain resources or not, based on the agent's attributes. [31]

In Figure 7, the Solid Architecture is illustrated. Solid applications obtain the user's WebID from a client certificate, provided by the browser. Using the WebID, the Profile Document can be located, which in turn links to the user's data pod. [5] Through the WebID-TLS protocol [32], which relies on public key authentication, Solid users can then authenticate against their personal data store via their client certificate. A pod can perform authentication by checking if the user, described by the Profile Document, is the owner of the private key that corresponds to the public key mentioned in the certificate. Alternatively, the WebID-OIDC protocol [33] derives the WebID from an OpenID Connect (OIDC) ID Token.

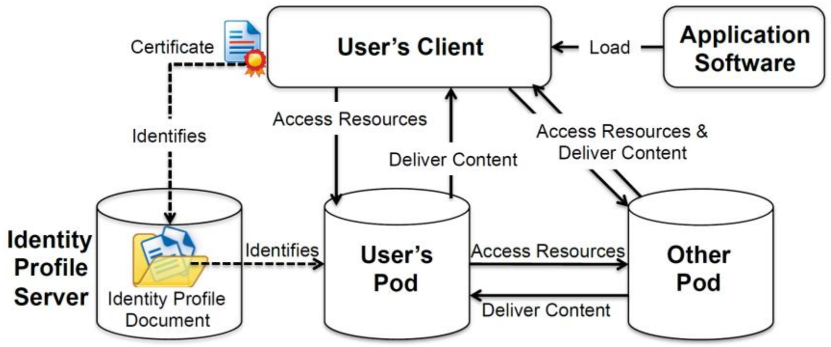


Figure 7 - The Solid Architecture. Source: [5]

2.5.1.6 Web Access Control

Access control at the level of resources and containers is enforced by the Web Access Control (WAC) system. Agents and applications can be allowed to read, write, control or append to resources. These are the four different access modes. Each resource or container can have a corresponding Access Control List resource (ACL), which contains a list of authorization statements. These rules define the agents that may access the resource, together with their access modes. ACLs can either be explicitly set or inherited from the resource's parent container. The URI of an ACL resource can be derived from the response of an HTTP GET or HEAD request. [5]

2.6 Querying over Linked Data

The difficulty with decentralized environments using Linked Data is data retrieval. Especially in use cases like decentralized social Web applications, query processing becomes a challenge as the number of independent data sources can increase quickly. Currently there are two main approaches [34], [35] for data integration and querying in a decentralized environment:

- Warehousing or materialization-based approaches. This method assumes a single, central database, in which RDF dumps of all remote data sources are collected in advance. By applying preprocessing and indexing techniques on the aggregated data, queries can easily be answered using the central repository.
- Distributed query processing (DQP) or federated querying. This approach does not need a central database, but instead queries the data sources directly. To that end, input queries are parsed and split into separate subqueries, which are then sent to the individual data sources, according to source selection. Finally, the responses from the remote sources are combined into a valid result.

2.6.1 Data warehousing

Because data warehousing offers the fastest query response times as a result of the elaborate preprocessing steps and the lack of need for network communication, it is momentarily the most widely adopted technique. However, this approach does have some flaws. Due to the collecting and indexing of RDF dumps from many different sources, which takes a lot of time, the data in the central repository is not always synchronized with its sources. Furthermore, data providers cannot control access to their resources, as queries are not addressed directly to the individual sources. Finally, most queries require only a small portion of the available data, so a lot of data aggregation, preprocessing and storage resources are unnecessary for individual queries. [34], [35]

2.6.2 Distributed Query Processing

As opposed to materialization-based approaches, distributed query processing works in a decentralized manner. Queries are performed against a *federation* of remote data sources. These sources individually respond to subqueries, after which a *federator* combines the results. For the user or application entering the input query (also called the consumer), it seems like one solid RDF store is being queried. To this end, federated querying is sometimes called *virtual integration*. [34]

Distributed query processing is advantageous in various ways. Data synchronization is inapplicable, as there is only one copy of the data and queries are issued directly against the original sources. As such, gathered resources are always up-to-date and data freshness is guaranteed. Moreover, adding or removing data sources does not cause any problems or take much time, as elaborate preprocessing and indexing processes are not required. This makes the DQP approach more flexible. Lastly, applications building a query do not need storage space for any central repositories, nor processing power, as this lays at the remote data sources. As such, query processing can be parallelized across all sources contributing to the query result. [35], [36]

Unfortunately, availability and reliability of this federated approach cannot be guaranteed because the system relies on a lot of possibly unstable or inactive data stores. [35]

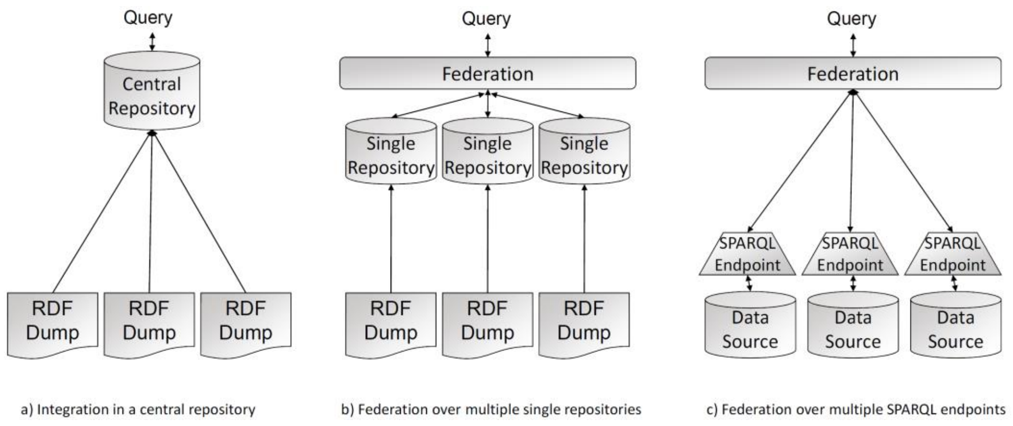


Figure 8 - Querying over distributed data sources. Source: [34]

Within the DQP approach, we can distinguish two variants, as is illustrated in Figure 8. The first diagram represents the data warehousing method, whereas the second and third diagram depict the two federation techniques.

In scenario b, the RDF dump from each data source is loaded into a separate repository. Either the data provider or the federator can populate these repositories. The federation layer splits the input query and aggregates the results from each repository. Due to the populating of the databases, there is still an overhead, but it stays limited as this process can be done in advance and so the federator can perform query optimization. However, write operations in this scheme are not straightforward, as they need to be propagated back to the source in some way.

Scenario c takes on a different approach. It does not depend on any repositories, but instead queries the SPARQL endpoints of the distributed data sources, again using a federation layer. As there is no populating of databases to be done, this setup presents no additional overhead next to the query execution process itself. However, as opposed to the previous scheme, in which the federator communicates with the separate repositories using their native API, this method requires the federator to conform to the rules of the SPARQL endpoints. These endpoints provide a more restricted access, and thus query processing is more challenging. [34] Moreover, the availability of SPARQL endpoints is problematic. [37]

Systems such as DARQ [23] and FedX [38] are existing solutions that implement the approach of scenario c. The workings of these systems are entirely transparent to the user, as the query processing mechanism over a federation of endpoints, and the dynamic addition and removal of endpoints in the network, happens behind the scenes. Both systems also introduced query optimization techniques, significantly improving query performance.

More recently, a third variant within distributed query processing, called Linked Data Fragments [37], was introduced. This approach aims to find the golden mean between the two extremes of data dumps and SPARQL endpoints. This is done by redistributing the load between clients and servers.

2.6.3 Query execution plan

According to research [35], generating an efficient query execution plan from an input query typically takes 7 steps:

1. Parsing the query into an internal representation.
2. Normalizing the query by applying equivalence rules.
3. Unnesting and simplifying of the logical query plan.
4. Optimizing by reordering the query operators.

5. Selecting sources that contribute (partial) answers.
6. Generating a physical query plan (by replacing the logical query operators with specific algorithms and access methods).
7. Executing the physical query plan.

In this query processing procedure, the most important problems are source selection and parallel data retrieval from the correct sources, in a way that these servers are not overloaded with requests. In the next section, we will look at some existing methods for finding sources that can contribute to a query.

2.6.4 Source selection

Most distributed query processing approaches are not very scalable when the network of data sources gains in size. Existing methods are often designed to handle only a small number of endpoints exposing a fairly large amount of data, instead of thousands of separate sources, each containing fewer resources. The latter case, however, would be ideal for decentralized social Web applications, whereby possibly thousands or millions of users, located all around the world, each have their own personal data pod. But when there is a very large number of data sources, checking every source for potential data contributing to the result of a query is unattainable. To that end, a source selection process must be executed, in order to rule out sources that do not contain valuable data for the query and keep only those that can contribute to the query answer. This process takes a triple pattern as input and returns a set of data sources that contain potential variable bindings. [35]

As source selection is the main difficulty with handling networks of large scale, a lot of research has been done to find more efficient methods, and some different approaches have already been developed. Some of these disparate techniques will be discussed in the following sections. They each have their own ideal use case and they are all methods that do not compute full indexes in advance, but rather do ad hoc query processing and possibly generate concise indexes. As such, these methods are fit for environments that do not have many resources or much processing power available. Not being able to perform extensive indexing operations in advance, however, does come with the price of longer query response times. If faster lookup times are desired, then materialization-based approaches to data retrieval over distributed sources are more fit for the job. [35]

2.6.4.1 Direct lookup

The most simplistic method is direct lookup (DL), which does not use an index structure, but rather looks up the resource URIs stated in the basic graph pattern of the query directly. The source URIs can thereby be deduced from the resource URIs in the basic graph pattern. As a result, URIs outside of the directly associated sources are left out. To that end, only a few types of triple patterns are supported by this approach and complete answers to queries are not always guaranteed. In addition, the direct lookup method visits the relevant data sources sequentially, as it encounters new sources during the query process. As opposed to that, other source selection techniques can work in parallel, simultaneously retrieving data from multiple servers. [35]

2.6.4.2 Schema-level Indexing

Information in the Semantic Web complies to schemas, and multiple resources can conform to the same schema. Schema-level indexing (SLI) exploits this by creating an index incorporating the schema. With this index, query processors can find out which data sources contain which predicates or classes. Consequently, queries of which the triple patterns contain a URI as predicate, or of which the patterns search for certain classes (using the `rdf:type` predicate), can be answered efficiently. Other types of queries are unfortunately not supported with this approach.

The schema-level index scales proportionally to the number of data stores and to the number of URIs they contain. As such, an SLI can get harder to maintain in larger distributed networks. [35], [36]

2.6.4.3 Inverted URI Indexing

An inverted URI index (II) specifies which URIs occur in which data sources. That way, given a resource URI, a query processor can use the index to determine a list of sources that store triples containing the URI, and thus may possibly contribute to the query response. This approach, as opposed to the previous two, accepts all types of triple patterns in the query's basic graph pattern. However, just as the schema-level index, an inverted URI index grows with the size of the network and the number of URIs. Therefore, the worst-case complexity of these two approaches is the same as when keeping a full index. [35]

2.6.4.4 Data summaries

An effective data structure to enable efficient and complete aggregation over many sources are data summaries [28], [35]. These lightweight summaries provide a succinct description of the contents of all available sources in a network, and as such, help the query processor decide how relevant a certain source is for a specific query. By only querying these relevant sources, query response times are reduced considerably. Not only do summaries simplify source selection, but they also help in query optimization by adding ranking capabilities and guiding in ordering join operations. In addition, data summaries are suitable for situations wherein direct lookup is desired, such as social networking applications, as they always provide up-to-date information when constructed at the right times.

Data summaries offer an approximate description of the data stored in the various sources, as opposed to schema-level indexes and inverted URI indexes, which are exact indexes. Summaries only naturally grow in size when the number of data sources increases, not necessarily when these sources store more URIs. Furthermore, summaries can be compressed to save space. The amount of compression can be chosen depending on how much space is available. Compression, however, is a matter of give and take, as the list of sources determined by the source selection process becomes less accurate, the more compact the summary.

As summaries represent the sources' data more broadly than other indexes, the source selection process can rule out more sources, and thus the returned list of sources is smaller. This results in faster execution times as there are fewer servers to be contacted.

To speed up the workings of data summaries, the RDF triples stored in the sources are transformed into numerical triples. This is done by applying hash functions to the subject, predicate, and object of the triples separately. The URIs of the sources that contain these triples are then inserted into lists of sources in the summary, according to the numerical ranges of the triples. With this approach, the query processor can first decide which range of numerical triples can answer the query, and then acquire a list of data sources by searching these triples in the summary. Some of these sources, however, may not contain relevant data for the query. This is because summaries are probabilistic, which means that they cannot produce false negatives, but they may produce false positives.

The two proposed approaches of data summaries in [35] are multidimensional histograms (MDH) and QTrees (QT). The MDH method ensures the construction and maintenance of the index to be efficient, at the cost of a lower accuracy. QTrees, on the other hand, provide a more accurate index, but building and maintaining it is more expensive. Therefore, the choice between MDH and QT is a trade-off.

Earlier research [28] embedded data summaries in Solid environments, with the main purpose of extending them to work in a privacy-preserving manner, by making sure unauthorized agents do not have access to restricted information. In addition, summaries can provide an efficient way to aggregate data from a large number of personal data stores, by enabling source selection.

The study suggested that each data pod may store several files, containing lists of RDF triples, and that one data summary is generated for each file, by the pod that stores the file. A possible third-party aggregator can then merge all these summaries into a combined summary, which contains approximate descriptions of all available data in the network of pods. This independent aggregator can be located on a separate server and *crawls* from source to source to be able to include all sources in its summaries. Possibly more than one aggregator can be engaged, which results in separate combined summaries for different source ranges. To that end, each aggregator must also maintain a list of all sources that were summarized by it. By utilizing combined summaries, client-side query engines can perform source selection during the querying process, and thus speed up query execution. The data summary approach in the Solid ecosystem, as envisioned by [28], is illustrated in Figure 9.

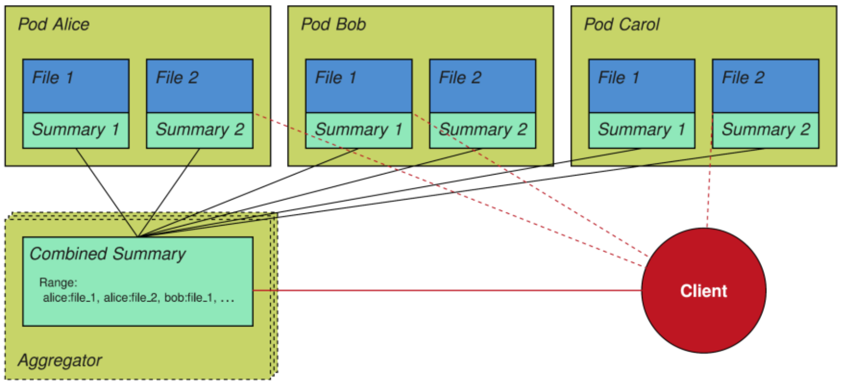


Figure 9 - Engaging data summaries in a Solid environment. Source: [28]

Aggregators and pods must keep their summaries up-to-date such that clients can correctly select relevant sources for their queries. If data summaries do not have the latest information on the files they describe, false negatives become a possibility. Data pods can update the summary of a file either by instantly regenerating it when the file's contents have changed, or by creating a new summary periodically. Aggregators can also reconstruct their combined summaries in a periodic fashion, or they can be notified by pods when the files they store have changed. [28]

While combined data summaries provide clients with relevant pods for their queries, the pods themselves manage access to the data they store. They do this by checking the access control policies for each file separately. These rules indicate which agents can access which parts of the data. Furthermore, as mentioned earlier, summaries can also be enhanced by adding privacy-preserving capabilities to them. [28]

2.6.4.5 Approximate Membership Functions

Approximate Membership Functions (AMFs) [28], [39], [40] are space-efficient, probabilistic data structures which enable one to efficiently test the membership of a specific element in a collection. The fact that they are probabilistic means that there cannot be any false negatives when testing the presence of items, but there can be false positives. For this, a fixed false positive probability can be set, influencing the size of the AMF. Consequently, AMFs involve a trade-off between size and accuracy.

AMFs have been used in related studies [39], [40] to enable more efficient querying over Triple Pattern Fragments (TPF) interfaces by reducing the number of HTTP requests performed by the querying process. They are particularly useful as a pre-filtering functionality, as they are generally much smaller than complete datasets. Other use cases of AMFs include the improvement of join operations in RDF environments [41] and the extension of the SPARQL `ASK` query form. [42]

Two different implementations of Approximate Membership Functions are Bloom filters [43] and Golomb-coded sets (GCS) [44]. A Bloom filter is a bitmap of size m, in which elements are inserted using k different hash functions. Initially, every bit is set to 0. When adding an element to the filter, the k hash functions produce k locations which must be set to 1 (using a bit-wise OR). Checking if an item is a member of the filter is done by applying the same k hash functions to the item and doing a bit-wise AND with the corresponding bits in the Bloom filter. If the result is false, the element is definitely not in the filter. If it is true, the element may possibly be in the filter. The chances of the element being member of the set depends on the false positive rate. If n is the number of items that must be added to the Bloom filter, and p is the desired false positive probability, then the filter size is $m = -\frac{n \ln p}{(\ln 2)^2}$. The optimal number of hash functions is $k = \frac{m}{n} \ln 2$. In situations where Bloom filters must be sent over the network or where space-efficiency is important, the filters may be compressed at the cost of extra delays for compression and decompression.

Golomb-coded sets are a variation of compressed Bloom filters, which uses only one hash function. GCS introduce a small overhead in size and slower decompression compared to compressed Bloom filters but achieve a higher compression rate. Compared to regular Bloom filters, GCS are still more space-efficient.

3. Problem statement

To this day, large-scale social Web applications such as Facebook and Twitter adopt a centralized approach for collecting and storing their users' personal data. This method is beneficial for companies as many of them need this data to be able to provide their services, but it invokes a lot of problems concerning GDPR compliances. [24] In addition, users of centralized social Web applications cannot reuse their data across different services or move data between applications, forcing them to duplicate the same data for each service they want to use. [5]

In reaction to these problems, the Solid ecosystem [5] was introduced. It enables the development of decentralized social Web applications by letting users store their own data on their personal online data stores. This gives them complete control over their personal information and allows the reusage of data between social platforms. For example, using Solid, one can store a picture on his or her pod, and multiple decentralized social applications can fetch that picture from the pod. For this, the user must explicitly grant access to these applications for that specific picture. As a result, only those applications can access the picture.

Unfortunately, while Solid solves some existing problems, it also introduces new problems. The biggest issue remains the inefficiency of retrieving data stored on the possibly very large networks of pods. Logically, given a decentralized social network in the range of thousands or even millions of users, checking every single pod for specific data based on an input query results in an abundance of HTTP requests and is therefore unattainable. Because of this, large-scale decentralized social Web applications are still nonexistent.

In order to mitigate this inefficiency and reduce the number of HTTP requests, some sort of source selection can be performed during the querying process. Earlier studies [39], [40] already evaluated the impact of Approximate Membership Metadata on the performance of Triple Pattern Fragment (TPF) interfaces, and one study [28] implemented Approximate Membership Functions in the Solid ecosystem for privacy-preserving purposes.

This research applies Approximate Membership Functions to a Solid environment in order to evaluate their effect on query performance. The emphasis lies on scalability of decentralized networks of datasources, by simulating an expanding social network. With this study, we hope to get somewhat closer to making large-scale decentralized social Web applications reality.

3.1 Research questions and hypotheses

In this research, a decentralized network of datasources will be simulated with an increasing number of sources to resemble a growing social network. The goal is to improve query performance over this network by implementing Approximate Membership Functions. As such, various metrics will be measured during the execution of the experiments, including total query execution time, number of HTTP requests, client memory usage and client CPU load. These metrics will then be compared to the metrics derived from the same experiments, but without the use of AMFs. The overhead posed by the construction of the AMFs will also be taken into account when comparing the two methods. In addition, different false positive probabilities will be tested, in order to find the optimal balance between query performance, the time it takes to create the AMFs, and the disk space they require.

To aid the development of this investigation, the following research questions are defined:

Question 1: To what extent can Approximate Membership Functions reduce the number of HTTP requests when executing SPARQL queries over an expanding network of datasources?

Question 2: To what extent can Approximate Membership Functions reduce the total query execution time when executing SPARQL queries over an expanding network of datasources?

Question 3: To what extent can Approximate Membership Functions reduce the memory usage and CPU load at the client, when executing SPARQL queries over an expanding network of datasources?

To answer the above three questions, not only will the number of datasources be increased, but the effects of varying the false positive probability of the AMFs will also be examined. Furthermore, different SPARQL queries will be tested to produce more reliable results.

Question 4: Which false positive probability ensures the optimal balance between performance, AMF construction time, and disk space?

Question 5: What is the impact of applying AMFs on the performance of the complete querying process, including the construction of the filters?

To perform the experiments, a client-side SPARQL query engine E will be extended to implement source selection based on Approximate Membership Functions. This extended version of the query engine will be referred to as E'.

The following hypotheses are defined as the predicted answers to the research questions:

Hypothesis 1: As the number of datasources grows, the difference in the average number of HTTP requests between E and E' becomes bigger. Using E' in larger networks allows for more datasources to be ruled out due to source selection, resulting in fewer HTTP requests.

Hypothesis 2: The use of Approximate Membership Functions significantly reduces the average total query execution time, as a result of the reduction of HTTP requests. Again, the difference between E and E' increases with a growing amount of datasources, starting at a negligible difference for very small networks but evolving to a significant difference for networks of a larger scale.

Hypothesis 3: As with the previous two hypotheses, the difference between E and E' concerning client-side memory usage grows with the number of datasources. In large networks, using E, much more sources need to be contacted and kept in memory, resulting in a significant increase in memory usage. The client-side CPU load during the experiments using E', however, is increased slightly due to the source selection process.

Hypothesis 4: On the one hand, AMFs with lower false positive probabilities contribute to a more precise source selection process and therefore better query performance. On the other hand, lower false positive rates result in higher AMF construction times and require more disk space. As such, the optimal balance can be achieved by neither choosing very low or very high false positive probabilities, but by choosing something in between the two extremes.

Hypothesis 5: The application of AMFs poses an extra overhead to the querying process due to the need for constructing them, but small enough to still achieve a significantly better query performance with E' than using E. The difference in performance between querying with E and querying with E' (including the AMF construction overhead) gets more significant as more datasources are added to the network.

4. Methodology

This section explains how this research was accomplished. First, the implemented technologies and frameworks are introduced, followed by the SPARQL queries used in the experiments. Then, the algorithms that were considered and used in the implementation are presented. Next, we will take a look at the experimental setup and conclude with the encountered problems.

4.1 Implementation

4.1.1 Data generation

In order to simulate a social network to run experiments over, a lot of synthetic data had to be generated. For this, a decentralized version of the LDBC SNB Data Generator [45] was used.[2] The LDBC Social Network Benchmark (SNB) [46] is an initiative with the purpose of testing functionalities in systems working with graph data. It uses a social networking environment to accomplish this. The benchmark comes with a data generator, able to generate synthetic social network data of large scale. With this, a large RDF file in Turtle syntax can be generated containing the data of persons, forums, blogposts, comments, et cetera. Figure 10 illustrates the data model of this generated social network.

The decentralized version of the data generator splits the large Turtle file so that the data is spread across a multitude of files, each containing the data of one specific entity in the social network. More specifically, each file contains all the triples of which the subject is one specific URI, denoting an entity in the network. In this research, for the purpose of performance, only the files containing data of *persons* and *cities* were used, with a maximum of 3500 persons and the 1231 cities they live in.

[2] The implemented version of this tool for this research can be found at https://github.com/thomasdevriese/Experimental-Setup.

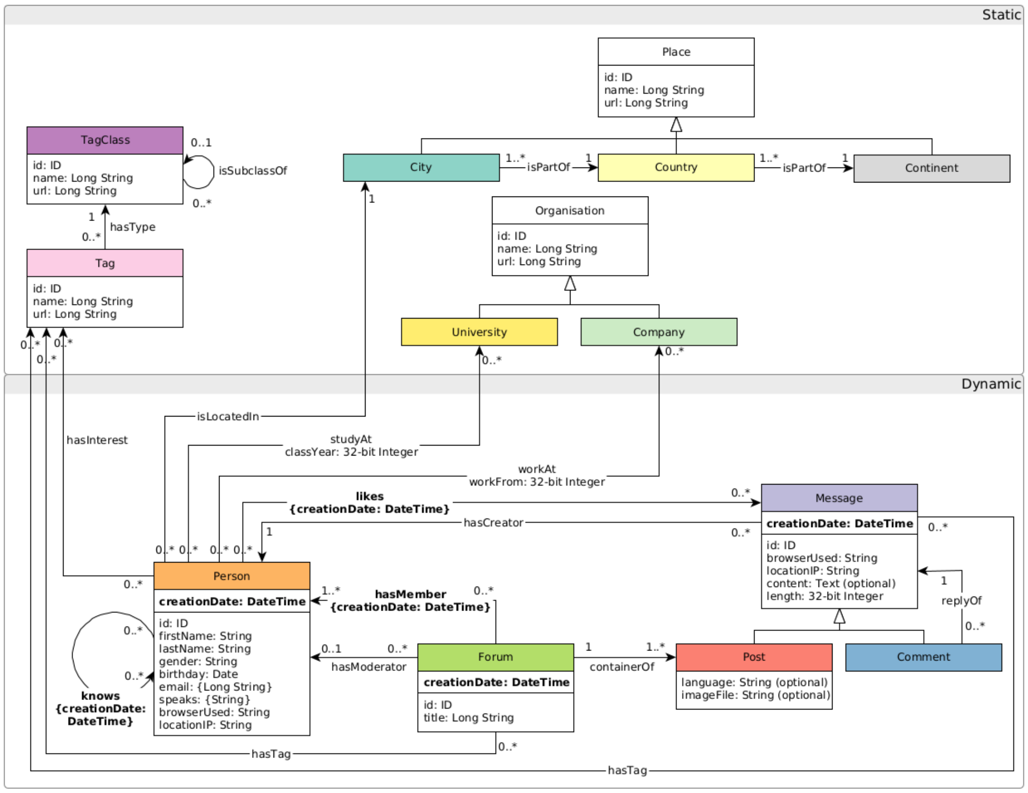


Figure 10 - Data model of the LDBC SNB Data Generator. Source: [46]

4.1.2 Server

Behind the scenes, the decentralized LDBC SNB Data Generator uses the Community Solid Server (CSS) [47] to host and serve the RDF files over HTTP. CSS is an open-source and modular implementation of the Solid specifications. Using the server, developers can create decentralized Solid applications and experiment with them. The decentralized data generator tool exploits the file-based store functionality of the Solid Server to serve the generated social network data over HTTP. That way, a decentralized social network can be simulated, as the data of every single entity must be accessed using a separate URI.

4.1.3 Approximate Membership Functions

For the construction of the Approximate Membership Functions, a separate script was written.[3] Part of the code in this script is a modification of an earlier implementation of AMFs, used in the Triple Pattern Fragments server. [48] The script uses Bloom filters as the probabilistic data structure. First, the number of bits in the bitmap of the filter and the number of hash functions

[3] The script for building Approximate Membership Functions can be found at https://github.com/thomasdevriese/AMF-builder.

are determined, based on the number of triples in the RDF file and the desired false positive probability. Then, a Bloom filter is initialized for each term (subject, predicate, object) by invoking an external npm package *Bloem* (which uses a bitbuffer internally) and passing the two parameters. Finally, the corresponding term of each triple is added to the filter. Listing 8 shows the AMF construction algorithm in pseudocode.

```
m = calculateBitmapSize()
k = calculateNumberOfHashes()
filters = ['subject','predicate','object']

for (variable in filters):
    filters[variable] = new Bloem(m, k)

for (triple in triples):
    for (variable in filters):
        filters[variable].add(Buffer.from(triple[variable]))

for (variable in filters):
    filters[variable] = {
        type: 'http://semweb.mmlab.be/ns/membership#BloomFilter',
        filter: filters[variable].bitfield.buffer.toString('base64'),
        m: m,
        k: k
    }
```

Listing 8 - Pseudocode of the AMF construction algorithm.

4.1.4 Client-side query engine

The most important aspect of the implementation of this research was the process of extending a client-side query engine to support source selection based on Approximate Membership Functions. For this, we used the Comunica [49] query engine. Comunica is a modular Web-based SPARQL query engine that allows for the development and testing of new Linked Data query processing functionalities. It enables federated querying over heterogeneous datasource interfaces out-of-the-box.

Comunica accomplishes its modularity by *wiring* different components together through dependency injection. Consequently, disparate query engine configurations can be initiated by employing different configuration files. Although each component operates separately from the other components, they have to be able to interact with each other in some way. For this, a set of software design patterns are put to use. The publish-subscribe pattern is implemented in combination with the actor model to allow for independent *actors* to execute different interpretations of a task and communicate with the *bus* responsible for that task. The mediator pattern is used to delegate actions to actors. The choice of which actor to designate for the execution of a task depends on the implementation of the mediator. It is also possible to combine the results of all actors subscribed to a bus.

The default configuration of Comunica follows a specific data flow upon an input query, of which a simplified version is illustrated in Figure 11.

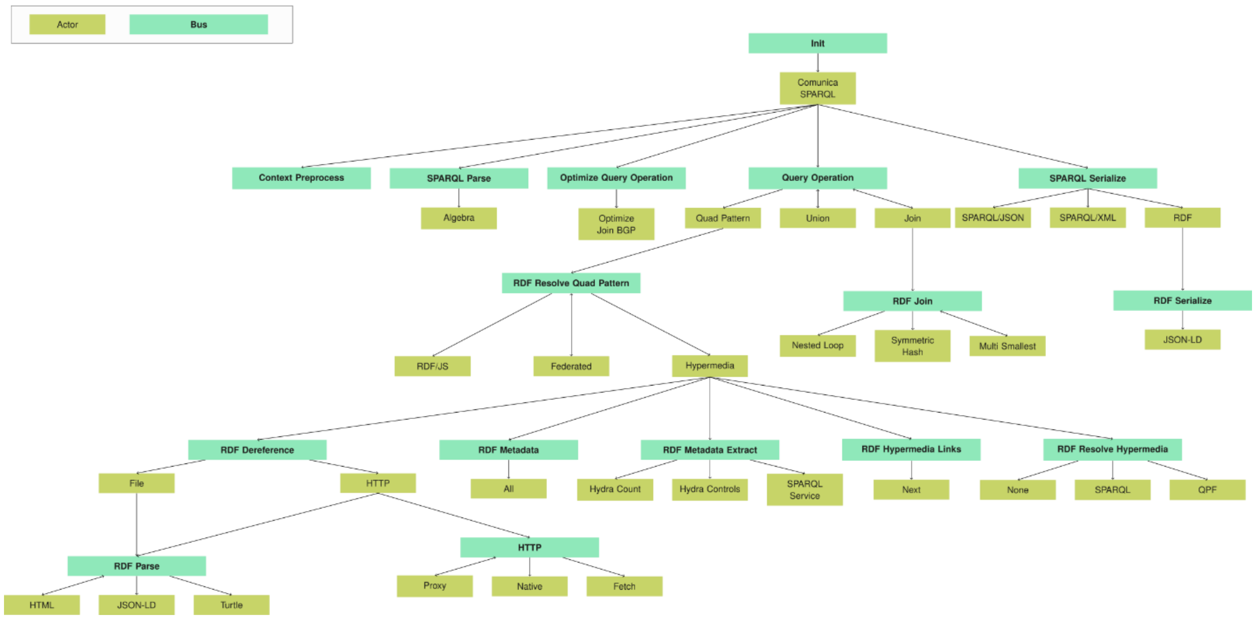


Figure 11 - Simplified data flow of the default Comunica configuration. Source: [50]

For this research, the modularity of Comunica was exploited to implement source selection capabilities based on AMFs. A new actor was developed[4] for the sole purpose of filtering the datasources array. This actor, called *Solid-Amf*, was positioned just before the *federated* actor and also subscribed to the *RDF Resolve Quad Pattern* bus. That way, when the data flow reaches the bus, it will first invoke the AMF actor to perform source selection, after which the *federated* actor is invoked. This actor will then continue the querying process using a limited sources array, based on source selection.

4.1.5 Source selection

The purpose of the generated Approximate Membership Functions is to enable a source selection process, in which many of the irrelevant datasources for a given query can be filtered out. This leaves a smaller set of sources to be queried over, which leads to better performance. The lower the chosen false positive probability for the AMFs, the smaller the filtered set of sources.

[4] The modified version of Comunica, including the new actor, is published at https://github.com/thomasdevriese/Comunica-AMF. The Solid-AMF actor can be found in the *packages* directory, under the name *actor-rdf-resolve-quad-pattern-solid-amf*.

Two slightly different algorithms for source selection were considered. The pseudocode of these algorithms is shown in Listing 9 and Listing 10. Ultimately, only the second algorithm was used in the experiments, as the first algorithm shows little to no reduction in the size of the filtered sources array. For example, if the input query contains a triple pattern `?s foaf:name "John"`, and the predicate `foaf:name` is found in a source's Bloom filter containing predicates, then that source will directly be added to the filtered set of sources, even though the subject `"John"` may not be present at that source. The problem is that a large number of sources contain the `foaf:name` predicate, causing almost every source to end up in the filtered sources array.

The second algorithm, however, checks the presence of all three terms in the Bloom filters. If one of the terms of the triple pattern is not found (and the term is no variable), the corresponding source is excluded from the filtered set of datasources. In most cases, this approach ensures a significant reduction of the size of the final sources array.

```
filteredSources = []
for source in originalSources:
    for term in ['subject','predicate','object']:
        if triplePattern[term] not variable:
            if source.bloomFilter[term].contains(triplePattern[term]):
                filteredSources.push(source)
                    break
```

Listing 9 - Pseudocode of the first source selection algorithm.

```
filteredSources = []
for source in originalSources:
    addSource = true
    for term in ['subject','predicate','object']:
        if triplePattern[term] not variable:
            if not source.bloomFilter[term].contains(triplePattern[term]):
                addSource = false
    if addSource:
        filteredSources.push(source)
```

Listing 10 - Pseudocode of the second source selection algorithm.

4.2 Experimental setup

To ensure the results to be as reliable as possible, the experiments were run with many different combinations of parameters. First, 15 different SPARQL queries were tested. These queries were constructed to fit the data model of the generated social network and were diversified enough to produce different results and query performances. We refer to Appendix A for a listing of the queries.

Next, the number of datasources in the social network was varied to obtain a network of 10, 100, 500, 1000, 2000 and 3500 sources. Note that for query 13 to 15, the total number of sources rose to a maximum of 4731, as those were queries which also searched for the cities in which the people in the social network live. The data of each city is contained by separate datasources. In a network of 3500 people, 1231 datasources are added (this is less for a smaller number of people), representing the different cities they live in. This brings the total number of datasources to 4731.

Thirdly, six different false positive probabilities were tested for the Bloom filters, being 1/4096, 1/1024, 1/128, 1/64, 1/4 and 1/2. Finally, each possible combination of the aforementioned parameters was iterated three times. Each time, the average of the metrics of those three iterations was calculated to obtain the definitive result.

In addition to the number of results produced by the query, four other metrics were measured, namely the number of HTTP requests issued by the query engine, the total query execution time, the client-side memory usage, and the client-side CPU load. To count the number of HTTP requests, an internal function of Comunica was used. For the query response time, the memory usage, and the CPU load, internal Nodejs functions were exploited. Unfortunately, the CPU load metric produced seemingly random values, making it unusable in the discussion of the experiment results. The memory usage metric showed some illogical values as well, but the main trend is still noticeable. Therefore, this metric is not excluded from the results in this work.

In order to make the experiment reproducible, a Bash script was written, which iteratively invokes the query engine with different parameters.[5] This way, the whole experiment could be run by executing only one script. A query timeout was set for 5 minutes, in combination with a maximum memory usage limit of 4096 MB for Nodejs. All experiments were run on a single machine with an Intel Core i7-8705G CPU at 3.10 GHz and 8 GB of RAM.

4.3 Encountered problems

Along the way of the implementation and execution of the experiments, some issues were encountered. The biggest problem that could not be solved was that the queries in the experiments had to be rather simplistic, as more elaborate queries (especially over a large number of sources) caused intensive memory usage and high query response times. This is likely due to the fact that bigger queries cause the query engine to perform more joins, which becomes

[5] The Bash script and the experiment results are published at https://github.com/thomasdevriese/Experimental-Setup.

problematic in an environment with many decentralized datasources. The LDBC SNB benchmark comes with a set of predefined queries, including rather simple ones, but these could still not be used in this setting. Therefore, we had to construct a set of new queries which were small enough to enable querying in a large, decentralized network, but still relevant enough to be useful in realistic scenarios.

Moreover, even with the smaller queries and the AMF-extended query engine, running the experiments in a network of more than 3500 datasources was unattainable. Some queries still produced results rather quickly but the majority of them caused poor performance or query failures.

The Community Solid Server also caused some issues during the experiments, but luckily, these could be solved. In situations with many requests to the server and high query response times, the server would throw consecutive *resource lock expired* errors. This issue was solved by increasing the *expiration time* of the *resourcelocker* utility. Another problem was the *ECONNRESET* error, which appeared at the client (but was probably invoked by the server) and was solved by setting the *keepAlive* parameter to false and increasing the *maxSockets* parameter of the HTTP-Native actor in Comunica.

Lastly, as mentioned earlier, the CPU load metric cannot be used in the results of this research, as no conclusions could be drawn from it.

5. Results

Various line charts and histograms were generated based on the results of the experiments.[6] The purpose of these data visualizations is to be able to provide reliable answers to our research questions. In the following sections, the obtained results will be discussed, and we will look back at the defined research questions and hypotheses.

5.1 HTTP requests

The first and most important metric to evaluate is the number of HTTP requests performed by the querying process. Figure 12 shows the number of HTTP requests per number of datasources, averaged over all queries (lower is better). The legend shows the different false positive probabilities of the Bloom filters (together forming E' from the hypotheses), whereby *Default* refers to the use of the query engine without Approximate Membership Functions (called E in the hypotheses).

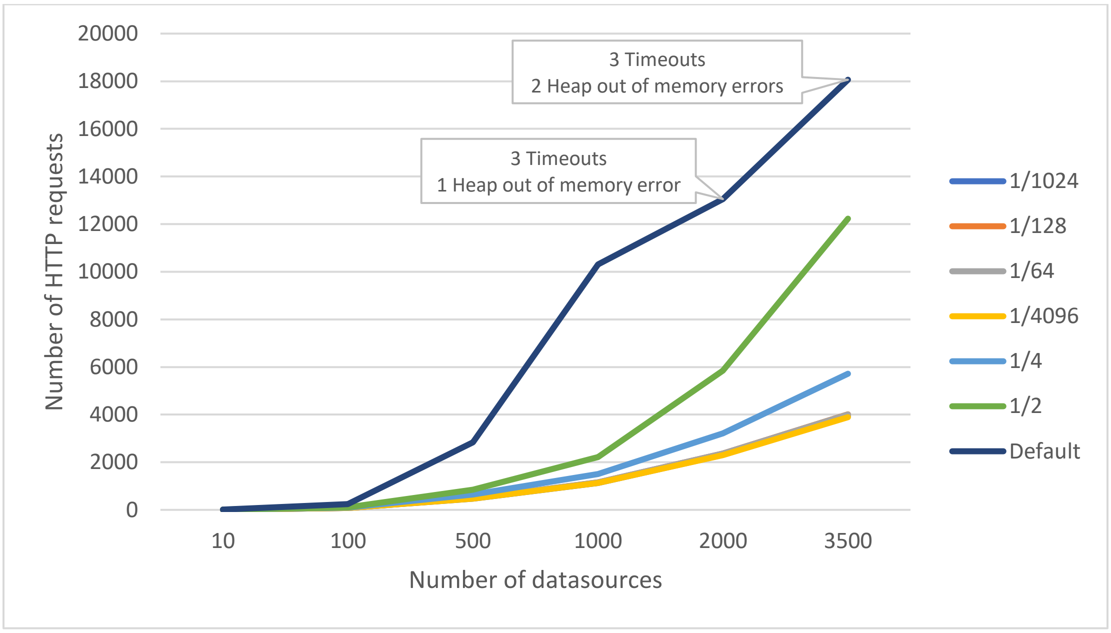


Figure 12 - Average number of HTTP requests per number of datasources.

[6] CSV files containing the results of the experiments can be found at https://github.com/thomasdevriese/Experimental-Setup.

Note that the chart is not completely accurate, as several queries (executed over 2000 and 3500 datasources) resulted in a timeout or a *heap out of memory* error. This only occurred in the default version of the query engine, not with the use of AMFs. These queries failed because the number of HTTP requests they perform is relatively high in comparison to other queries. However, due to these failures, no metrics have been measured and as such, the number of HTTP requests of these queries has not been included in the averages. Therefore, in reality, the averages for 2000 and 3500 datasources using the default version are much higher, following the pattern found in the lower numbers of datasources.

Still, the default version of the query engine immediately stands out, as the number of HTTP requests increases much quicker than with the use of AMFs. Going from 500 to 1000 datasources in the default version results in an increase of 264%, while the increase with the use of AMFs at a false positive rate $p = 1/64$ is just 137%. The difference in the average number of HTTP requests between E and E' (at $p = 1/64$) for 500 datasources is 2347 requests. The same difference for 1000 datasources is 9152 requests, which means an increase of 290%.

While the number of HTTP requests at $p = 1/4096$, $p = 1/1024$, $p = 1/128$ and $p = 1/64$ almost perfectly align, a significant increase is observed at rates of $p = 1/4$ and $p = 1/2$. At 3500 datasources, $p = 1/2$ performs more than 3 times as many HTTP requests in comparison to $p = 1/4096$. Therefore, high false positive probabilities contribute to poorer query performance.

To examine the results more into detail, some clustered column charts were constructed, one for each number of datasources. These histograms show the exact number of HTTP requests for each executed query (averaged over 3 iterations). Figure 13, Figure 14 and Figure 15 show the observations for 10, 1000 and 3500 datasources, respectively. It is clear that an increase in the number of datasources leads to a bigger difference between E and E', at least for low false positive rates. As such, **hypothesis 1** is validated.

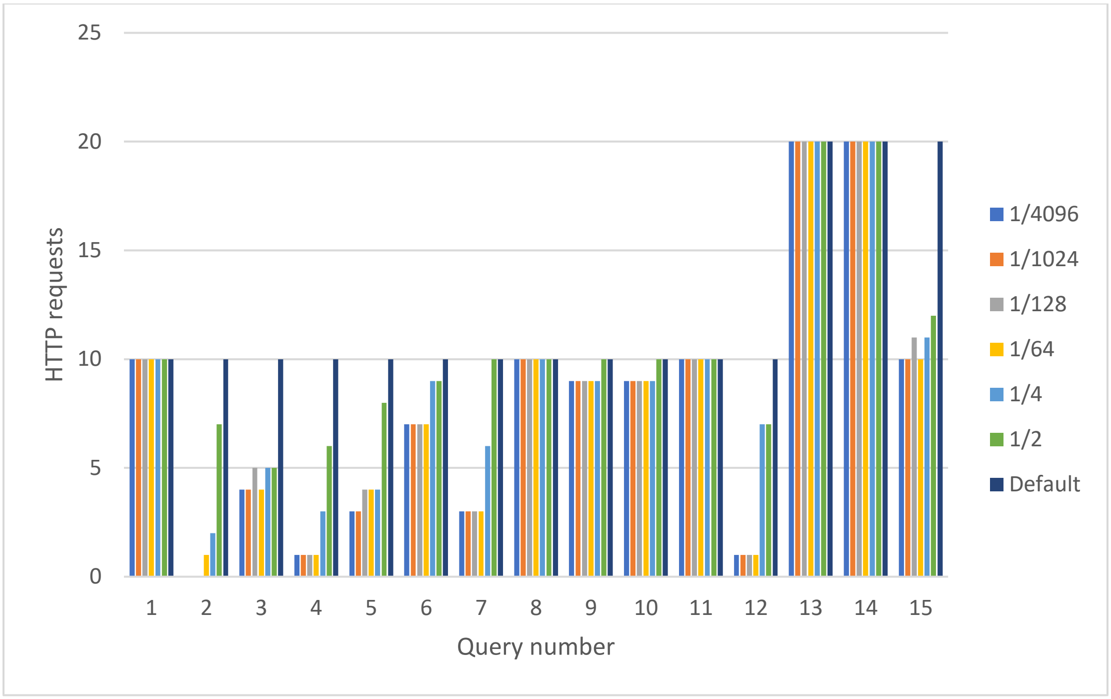


Figure 13 - Number of HTTP requests per query for 10 datasources.

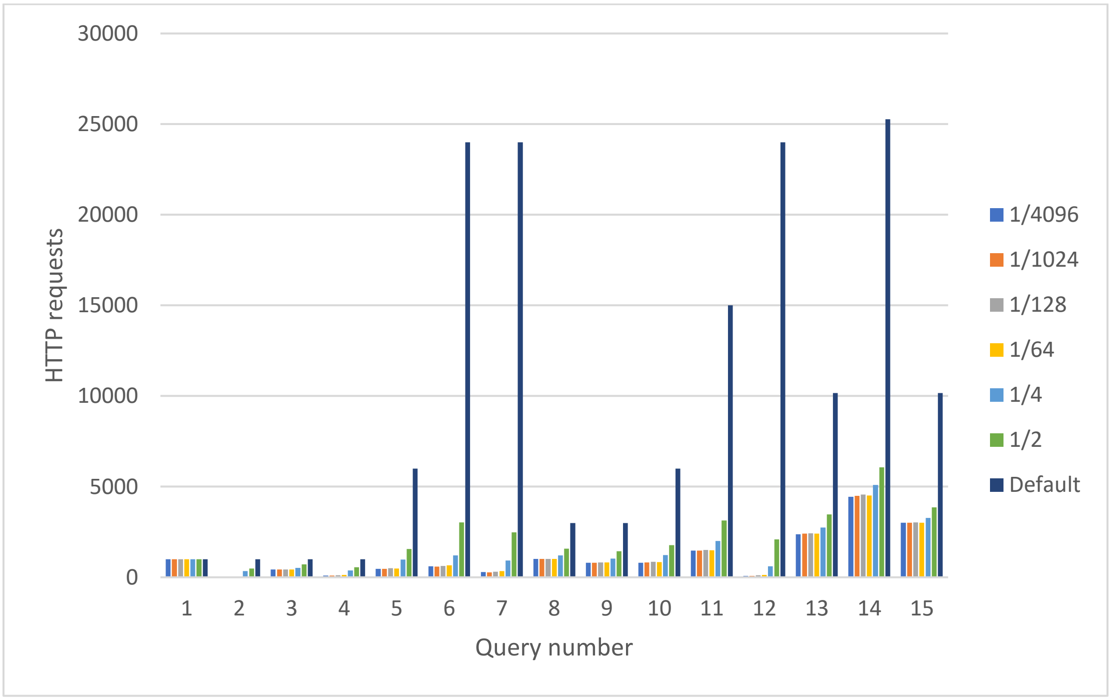


Figure 14 - Number of HTTP requests per query for 1000 datasources.

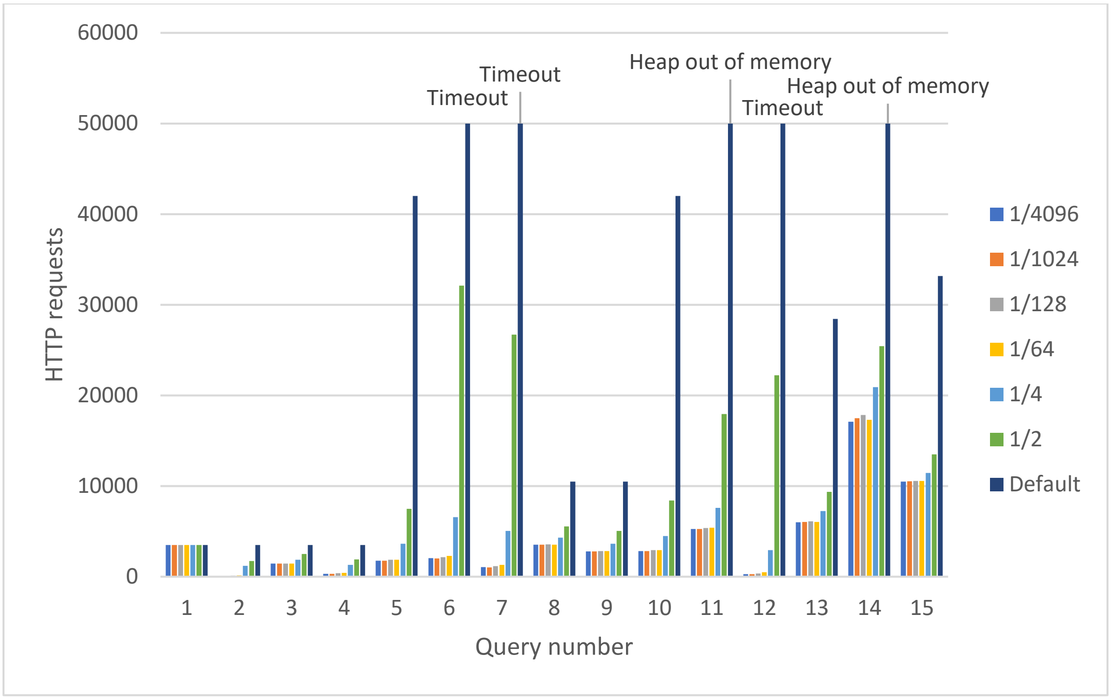


Figure 15 - Number of HTTP requests per query for 3500 datasources.

On Figure 15, it is indicated which specific queries failed due to timeouts or *heap out of memory* errors. For the purpose of visualizing the failures, the bars of these queries have been given a high value. We refer to Appendix B for the remaining charts, depicting the results for 100, 500 and 2000 datasources.

As a final example, Figure 16 shows the evolution of the number of HTTP requests for one specific query. This chart also clearly displays the effect of expanding the network of datasources, being that the number of HTTP requests rises much more quickly when using the default query engine than when using the AMF approach.

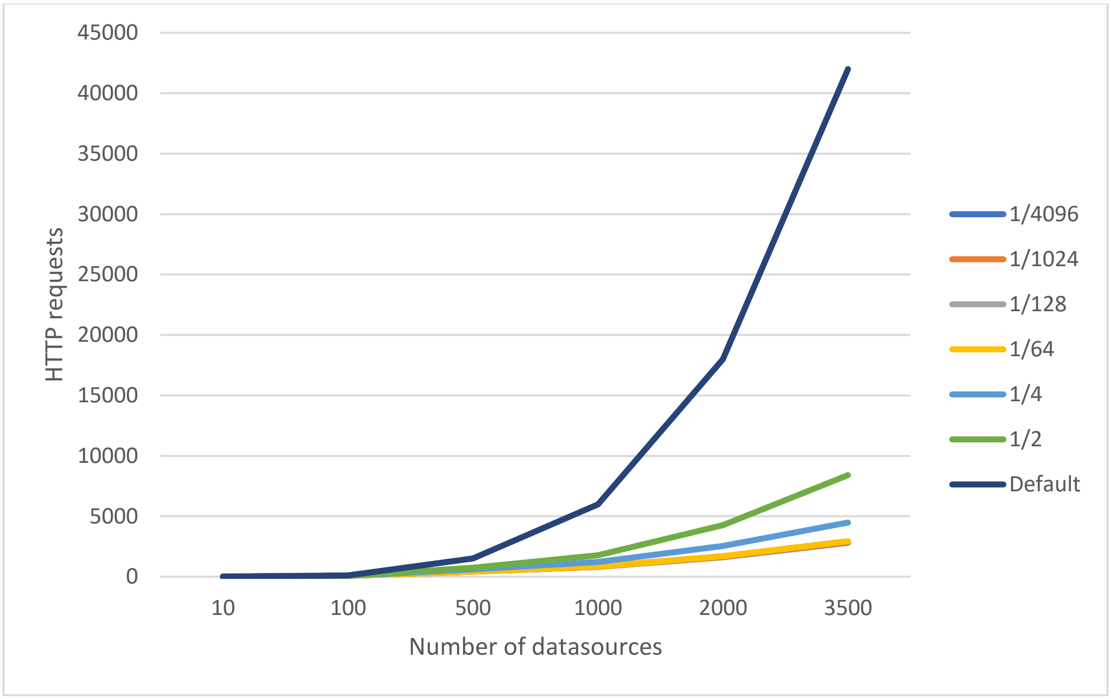


Figure 16 - Number of HTTP requests per number of datasources for query 10.

5.2 Query execution time

In addition to the number of HTTP requests, the total query response times were also measured. This metric is directly affected by the HTTP requests, as an important part of the querying process involves contacting remote datasources. Figure 17 illustrates the total query execution time per number of datasources, averaged over all queries (lower is better).

As with the HTTP requests, it must be noted that this chart is not completely accurate due to the query failures without the use of Approximate Membership Functions. However, in this case, it is known for certain that the timed-out queries have a query response time of at least 5 minutes, and some possibly a lot higher. To that end, for the purpose of obtaining a more correct visualization, the query execution times of the timed-out queries were manually set to 320 seconds (just above the maximum). The response times of the queries that resulted in a *heap out of memory* error, however, cannot be known and are therefore not included in the averages. Due to these failures, the line showing the averages obtained with the default query engine is again lower than it would be in reality.

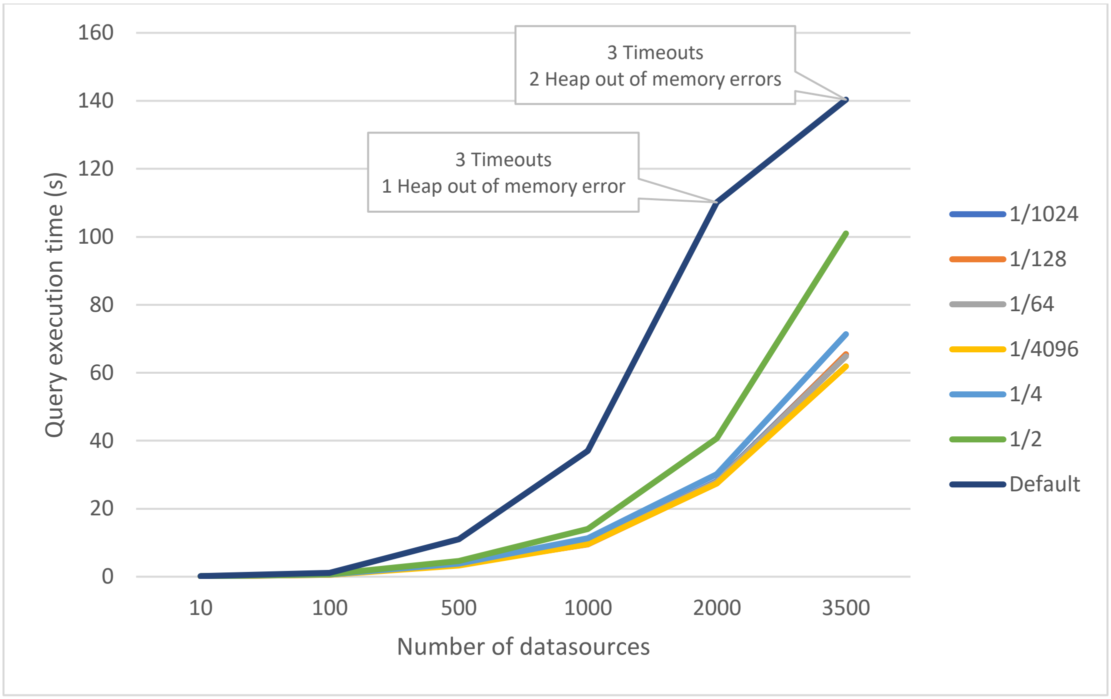


Figure 17 - Average query execution time per number of datasources.

The same pattern arises as with the number of HTTP requests. E' with low false positive probabilities shows the best response times, followed by E' with higher false positive rates. The averages of E are relatively close to those of E' for very small networks but increase much more rapidly when the network expands. Going from 500 to 1000 datasources while using E' and $p = 1/64$ produces an increase of 162% in the average query response time. This increase rises to 236% using the default query engine. The difference between E and E' (at $p = 1/64$) when scaling up from 500 to 1000 sources shows an increase of 273%.

False positive rates $p = 1/4$ and $p = 1/2$ again show reduced performance in comparison to lower rates. $p = 1/4096$ contributes to slightly better results than $p = 1/1024$, $p = 1/128$ and $p = 1/64$, of which the averages almost perfectly align.

Figure 18, Figure 19 and Figure 20 illustrate the query execution times for 10, 1000 and 3500 datasources more into detail through clustered column charts. As with the HTTP requests, we can observe that the difference between E and E' grows with the expansion of the network of datasources, validating **hypothesis 2**. Queries that resulted in timeouts or *heap out of memory* errors are indicated on Figure 20 using a label and a value slightly higher than the timeout value. The remaining charts can be found in Appendix C.

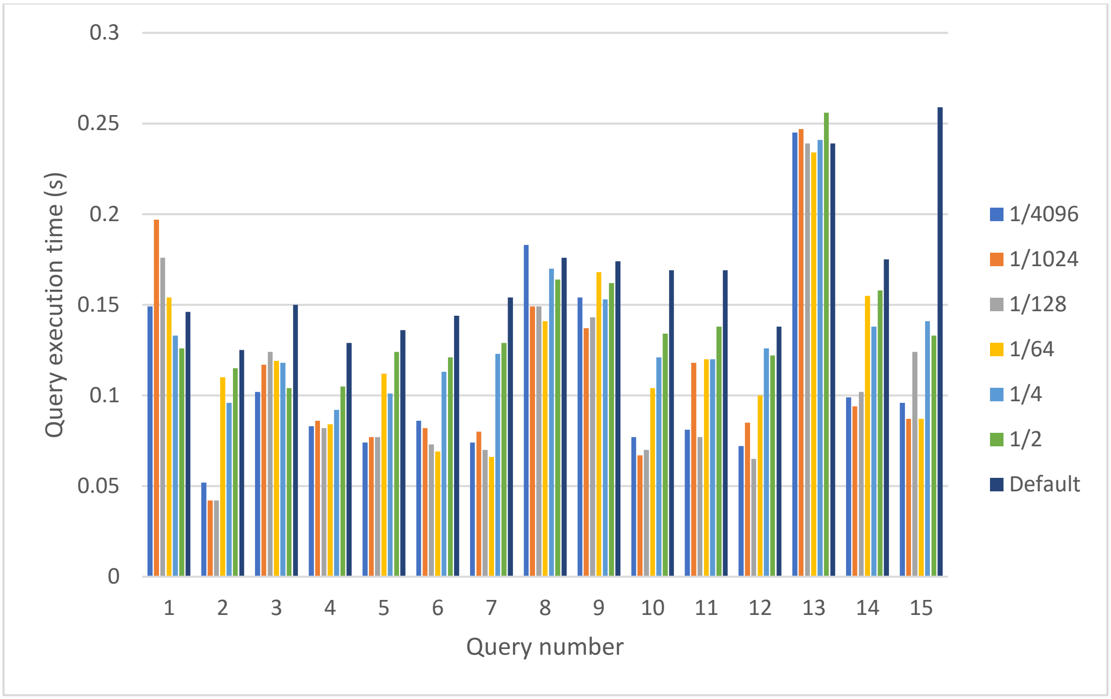


Figure 18 - Query response time per query for 10 datasources.

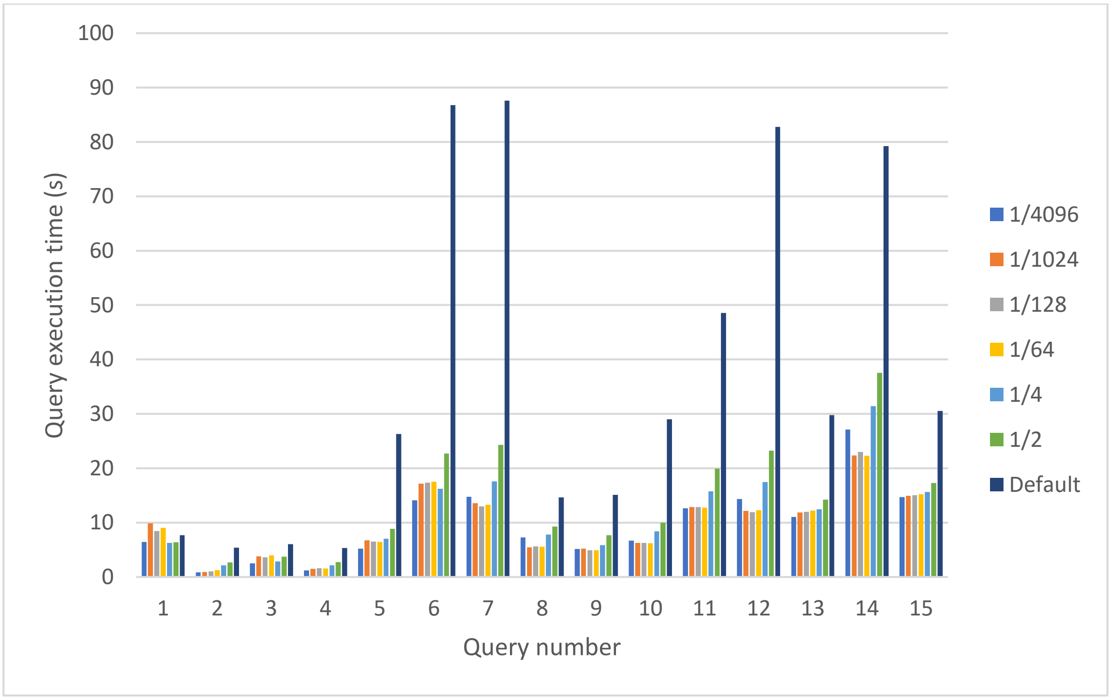


Figure 19 - Query response time per query for 1000 datasources.

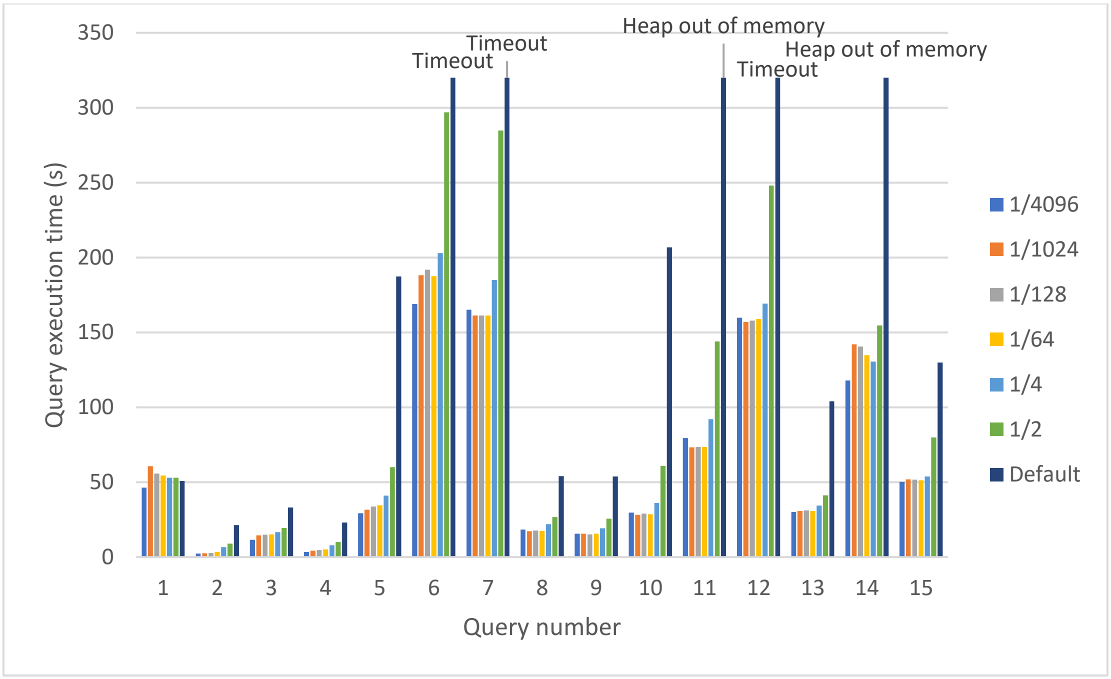


Figure 20 - Query response time per query for 3500 datasources.

The evolution of the response times for one specific query with an increasing amount of datasources is depicted in Figure 21. Not applying source selection results in much a faster increase of the response time. Again, the higher false positive rates perform worse than the lower rates.

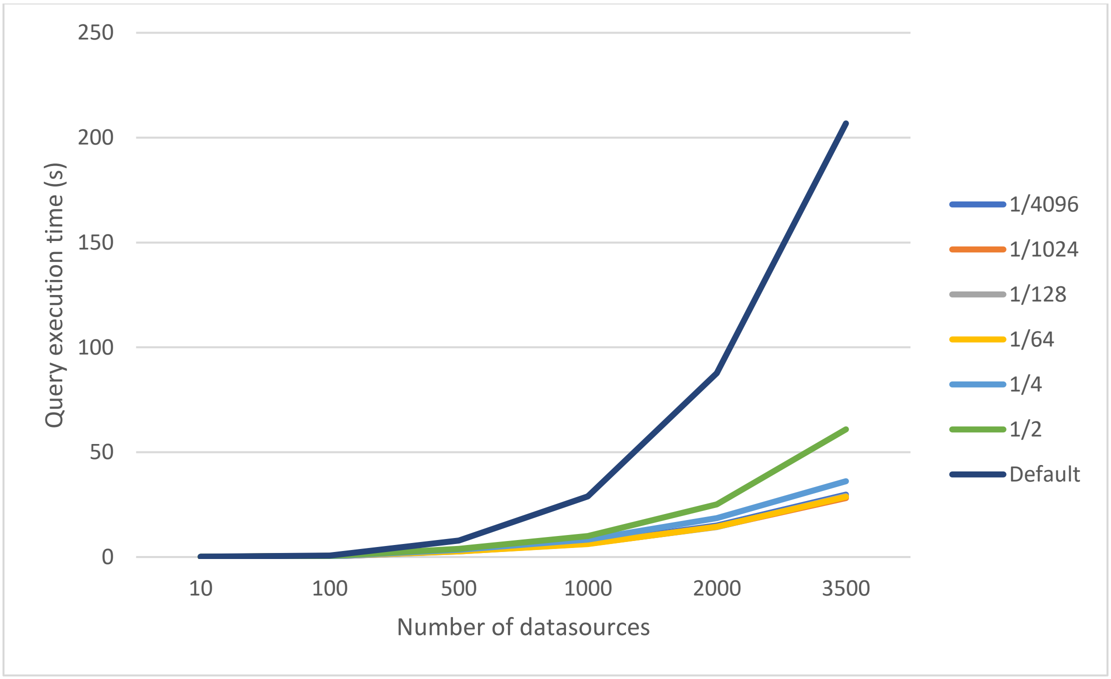


Figure 21 - Query execution time per number of datasources for query 10.

5.3 Memory usage

The third measured metric is the memory usage at the client, who runs the query engine. Just as the query response time, memory usage is also greatly affected by the number of HTTP requests. More sources to be retrieved means more data to keep in memory. Figure 22 displays the evolution of the memory usage with an increasing number of sources, averaged over all queries (lower is better).

Again, we must note that the chart is not completely accurate due to some query timeouts and failures. This time, it is known for certain that the queries which resulted in a *heap out of memory* error cause a memory usage of at least 4096 megabytes. Because of this, for the purpose of visualization, the memory usage of these queries was set to 5120 MB. The timed-out queries, however, are not included in the averages.

Another note to be made is that the memory usage metric in Nodejs is not as reliable as the other metrics discussed in this section. Some of the obtained values seem somewhat illogical, but the main trend is still clear.

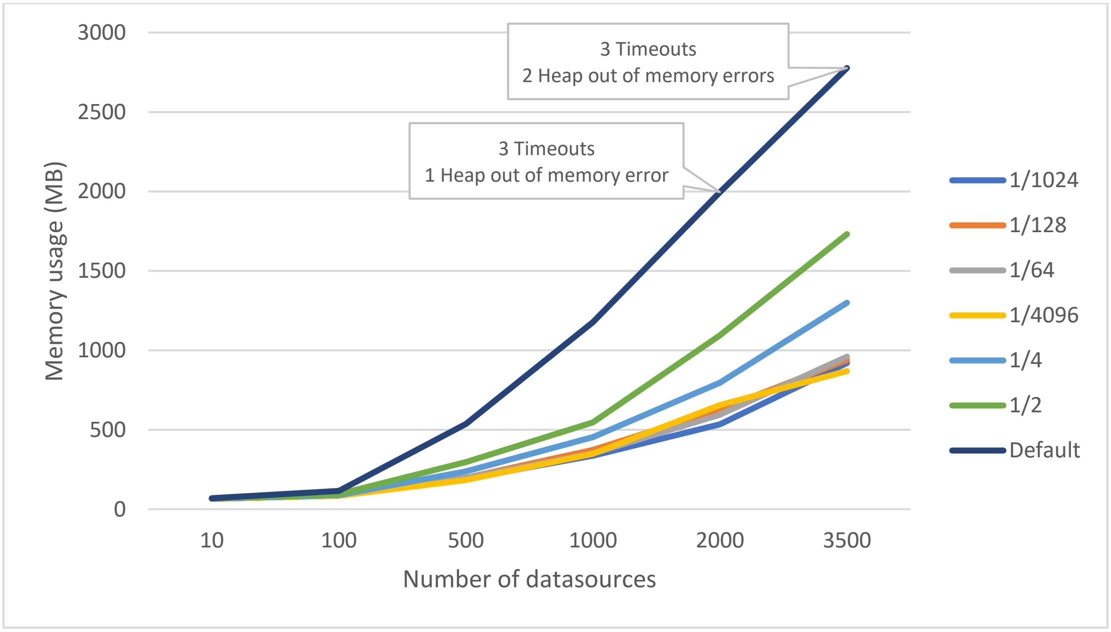


Figure 22 - Average memory usage per number of datasources.

As the number of HTTP requests has a big influence on the client memory usage, we can observe the same pattern here. Just as with the use of AMFs, the memory usage of the regular query engine setup starts off fairly low. But again, adding more datasources means a faster increase for E than for E'. Moving from 500 to 1000 datasources using the default engine and with $p = 1/64$ causes an increase of 78% in the average memory usage at the client. The same calculation for the default engine yields an increase of 119%. The difference between E and E' (at $p = 1/64$) when scaling up from 500 to 1000 data stores increases 143%. Therefore, **hypothesis 3** is partly validated. Unfortunately, we cannot make any statements concerning client-side CPU load, as this metric appears to consist mainly of arbitrary values.

Lower false positive rates again show better performance than $p = 1/4$ and $p = 1/2$. Oddly, the probability $p = 1/4096$ causes higher memory usage for 2000 datasources than $p = 1/1024$, $p = 1/128$ and $p = 1/64$, but then again lower usage for 3500 sources. This is likely due to the lower accuracy of the memory usage metric.

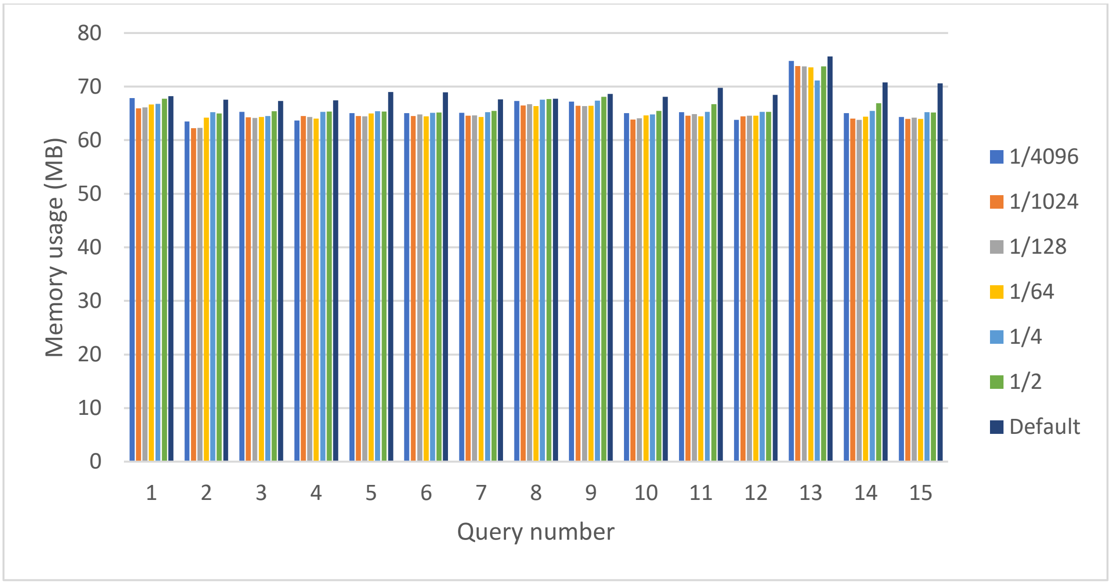


Figure 23 - Memory usage per query for 10 datasources.

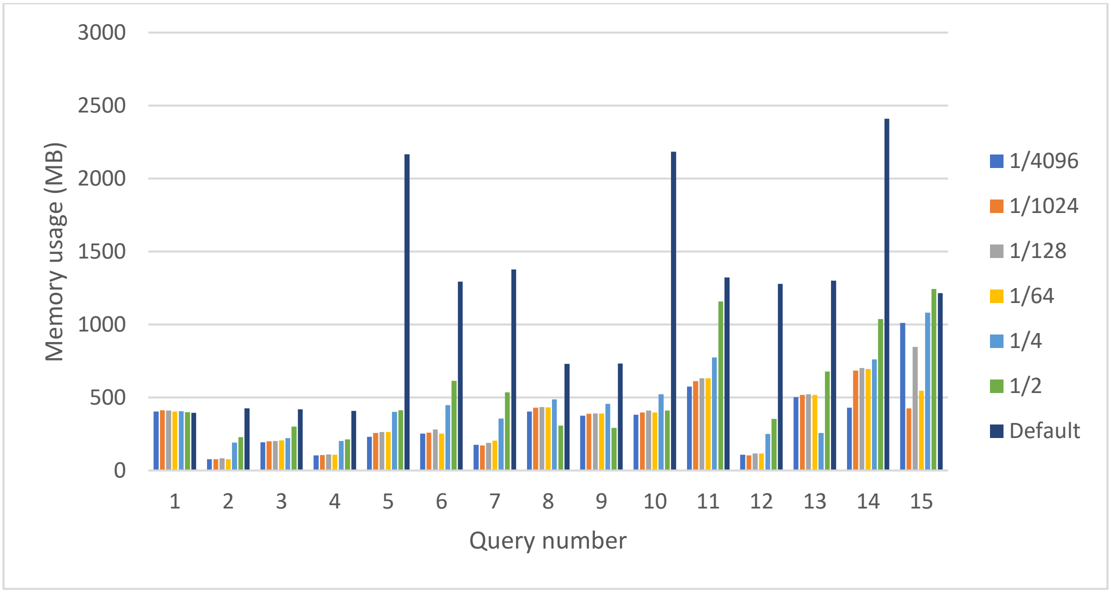


Figure 24 - Memory usage per query for 1000 datasources.

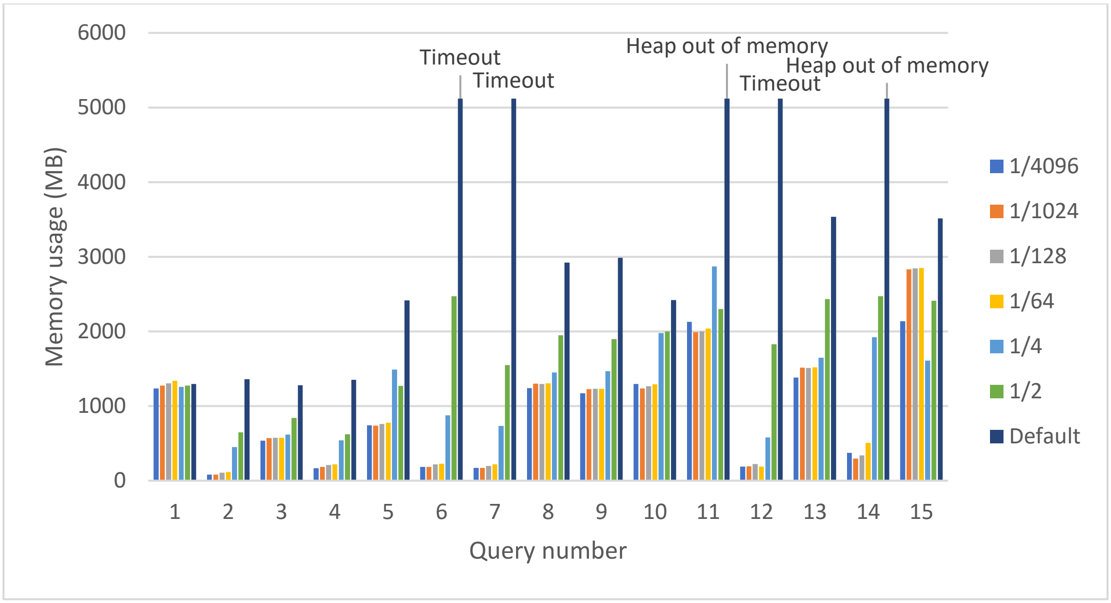


Figure 25 - Memory usage per query for 3500 datasources.

More detailed results for 10, 1000 and 3500 datasources are presented in Figure 23, Figure 24 and Figure 25, respectively. The more sources are added, the more the memory usage of the regular setup stands out again, following the pattern from the previous two sections. The clustered column charts for 100, 500 and 2000 datasources can be found in Appendix D.

Figure 26 depicts the progress of the client-side memory usage for one specific query. Although there are some illogical values due to the metric's inaccuracy, the difference between E and E' is still clearly visible.

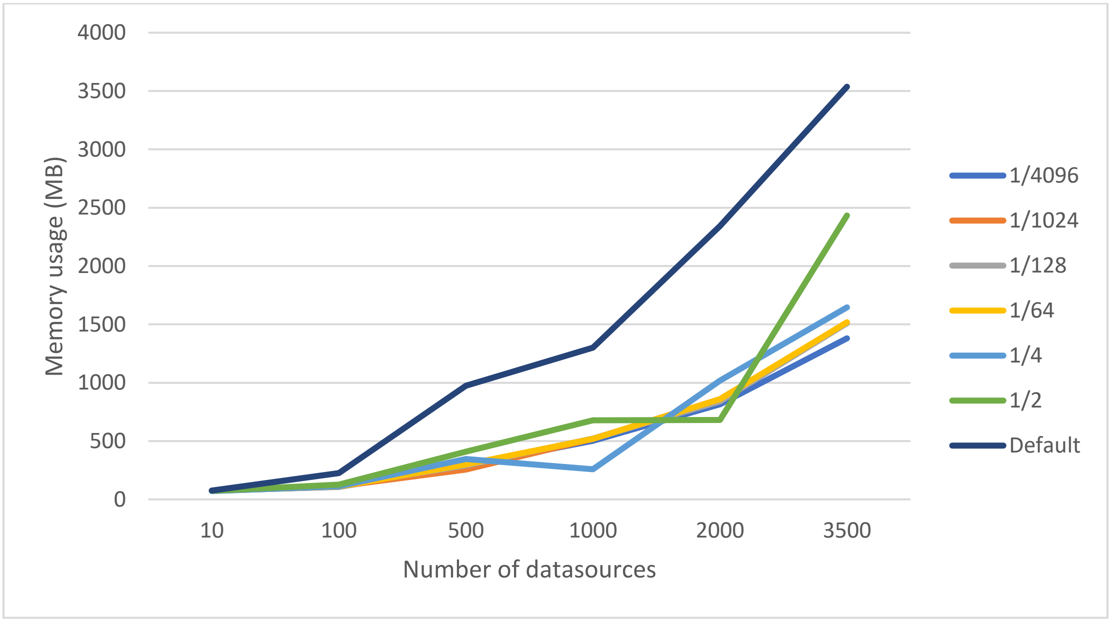


Figure 26 - Memory usage per number of datasources for query 13.

5.4 AMF construction

When comparing a query engine with or without the use of Approximate Membership Functions, we must not only evaluate the querying process itself, but also take the construction of the AMFs into account. After all, Bloom filters do not appear out of nowhere, they must be created at some time by some process.

To evaluate the performance of the AMF construction process, some experiments were carried out in which the time it takes to create the Bloom filters was measured, together with the disk space these filters require. As the experiments in the previous sections were executed over a maximum of 3500 datasources, each containing one person's info, the same 3500 sources are used in the experiments in this section. However, an additional 1231 datasources, each containing the data of a location, are included as well. This brings the total to 4731 datasources for the experiments concerning AMF construction.

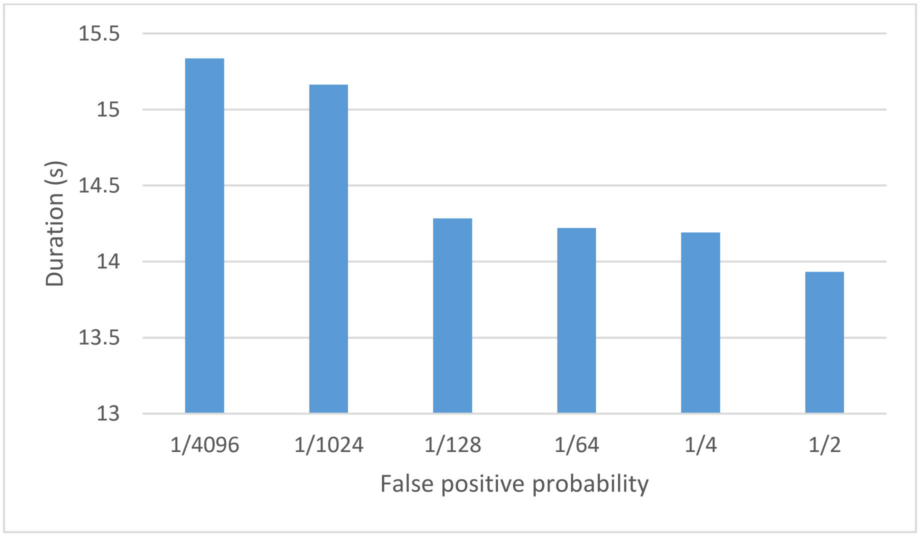


Figure 27 - Construction time per false positive probability for the Bloom filters of 4731 datasources.

Figure 27 reveals the construction time of the Bloom filters of 4731 datasources, per false positive probability. Note that for each datasource, 3 filters are created: one for the subjects, one for the predicates and one for the objects. Therefore, a total of 14.193 Bloom filters are created for each false positive rate.

The histogram clarifies that a lower false positive probability leads to a higher overall construction time, with a maximum of 15.3 seconds for $p = 1/4096$ (approximately 3 milliseconds per datasource). This behavior is expected, as a lower false positive rate means more precision, and so the filter must contain more data and thus, it takes more time to create. The difference between the two extremes is relatively small, being only 1.4 seconds. However, this number can quickly rise in networks with a scale of hundreds of thousands or even millions of datasources.

If we look at the disk space these Bloom filters require, the same pattern can be observed. This is illustrated in Figure 28. Adding more precision to the Bloom filters, requires them to contain more data and take up more space. Although the difference in construction time between the lowest and highest false positive rate is relatively small, the difference in size between these two extremes is a lot bigger. The filters with a probability of $p = 1/4096$ require almost 3 times as much space as the filters with $p = 1/2$. In this setting, that difference is only 2.37 MB, but in networks with very large scale, the difference can be crucial.

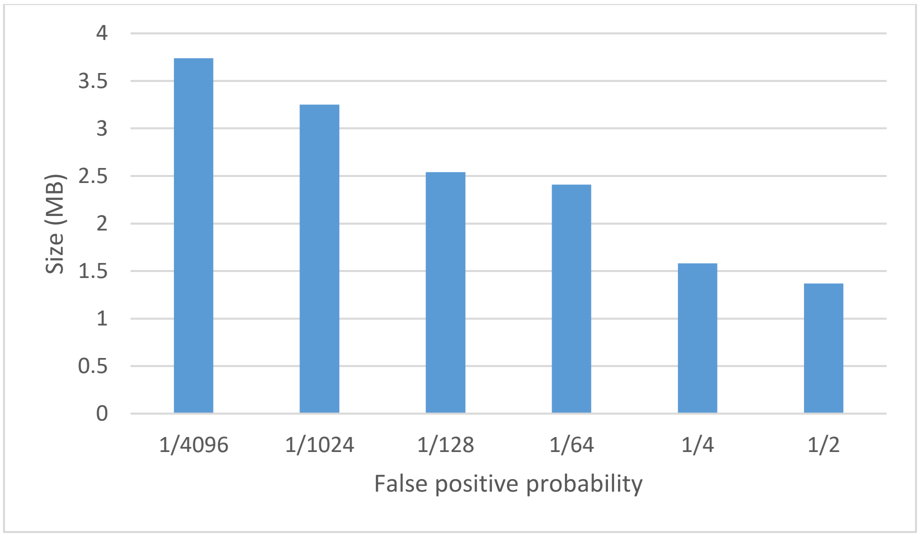


Figure 28 - Disk space per false positive probability for the Bloom filters of 4731 datasources.

In the previous sections, we learned that lower false positive rates do it better, performance-wise. Contrarily, in this section we discovered that Bloom filters with lower probabilities take more time to create and require more disk space. Therefore, the optimal balance between query performance, construction time and disk space lies somewhere between the highest and the lowest false positive probability. As such, **hypothesis 4** is validated. It seems $p = 1/64$ would be the recommended false positive rate, as it performs much better than $p = 1/4$ and $p = 1/2$, while less time and disk space are required to create the filters than with $p = 1/4096$, $p = 1/1024$ and $p = 1/128$ (and it barely differs from them performance-wise).

To answer **research question 5** (What is the impact of constructing AMFs on the performance of the complete querying process?), we have to take some things into consideration, such as when the AMFs are constructed, and which entity is constructing them. This will be discussed further in the next section.

6. Discussion

The use of Approximate Membership Functions when querying over an expanding network of datasources proves to be advantageous in multiple ways. The number of HTTP requests, as well as the overall query execution time and the client-side memory usage are improved significantly by using Bloom filters. We saw that the difference between the regular query engine and the AMF-extended query engine is negligible for very small networks with no more than 100 datasources, but as soon as that number increases, the difference increases with it. A relatively reasonable increase in the number of datasources from 500 to 1000, results in a critical increase of 273% in the difference in query response times between E and E' (at $p = 1/64$). If we follow this pattern beyond the tested 3500 datasources, we can expect to see the difference between E and E' growing further. With these results, research questions 1 to 3 are answered and their hypotheses validated (except for the client-side CPU load).

Constructing the Bloom filters for 4731 datasources takes a maximum of 15.3 seconds for $p = 1/4096$ and a minimum of 13.9 seconds for $p = 1/2$. As such, the difference in construction time is relatively small, at least for a network of around 5000 datasources. The required disk space for the combined Bloom filters, however, shows a more notable difference, with a maximum of 3.74 MB for $p = 1/4096$ and a minimum of 1.37 MB for $p = 1/2$. From these results, in combination with the results regarding query performance, we can deduce that the optimal false positive probability is $p = 1/64$. This false positive rate provides the optimal balance between performance, filter size, and filter construction time. Consequently, research question 4 is answered and its hypothesis validated.

Research question 5, however, is yet to be answered. As mentioned, some things apart from the experiment results must be considered when answering this question. The problem with the current method of applying Approximate Membership Functions is that, without the use of an external server taking care of the AMFs, the number of HTTP requests at the client can rise up to more than the number of HTTP requests when using the default query engine. This is because of the need of the Bloom filters to be constantly up to date. Given an input query, the client would first have to contact every datasource in the network in order to construct the filters. Only thereafter would the client be able to perform source selection and execute the query. As such, the total number of HTTP requests for each query would be equal to the sum of the number of datasources in the network and the number of sources retrieved from the source selection process. As a result, query performance would be worse than with the default setup.

Luckily, there are a few solutions to this problem, of which two will be discussed here. Both of them implement an aggregator, which is separated from the client. Both also have their own preferable use cases.

The first solution is to add an aggregator to the network, which crawls from datasource to datasource and keeps a list of all sources in the network. It can renew all AMFs periodically, or it may generate a new filter upon file changes, in which case the datasource can send a notification to the aggregator. When the client wants to perform a query, it can let the aggregator know, after which the aggregator combines all filters into one large file and sends it back to the client. Then, the client can perform source selection and execute the query. This method adds only one HTTP request to the complete process of the client. Moreover, this solution becomes particularly interesting if the client needs to execute multiple queries across the same range of sources. In that case, it can download the combined AMF file from the aggregator once and reuse it for different queries. However, this approach is not ideal for very large networks of sources, as the file containing the AMFs grows with the number of sources and must be sent to the client over the network. For $p = 1/64$ and 4731 sources, sending a file with a size of 2.41 MB is achievable. Scaling the network up to 100.000 sources, however, leads to a size of 51 MB, while a network of 1 million sources means an unattainable 510 MB.

The second method is more favorable to large-scaled networks. In this approach, the aggregator discovers new sources and updates its AMFs in the same way, but it differs in the querying process itself. When the client must execute a query, it first sends the query over to the aggregator. After that, the aggregator performs the source selection process itself, and sends the list of selected sources back to the client. Based on this list of sources, the client can immediately execute the query. Just as with the first solution, this approach adds only one HTTP request to the complete querying process of the client. In this method, the size of the network matters less, as the query and the list of sources are the only pieces of data to be exchanged with the aggregator. However, this solution demands more processing power from the aggregator. Furthermore, this method may be less ideal in situations where protection of privacy plays a very important role, as each query from the client must be sent over the network and can be read by the aggregator. Therefore, the choice of which solution to implement mostly depends on the use case, but this trade-off needs to be further investigated in future work.

With the proposal of these two solutions, **hypothesis 5** can also be validated. Applying Approximate Membership Functions does indeed pose an overhead due to its construction time, but as the aggregator takes on the task of creating the filters, the only noticeable overhead is the communication between client and aggregator. Therefore, the overall performance lies very close to the obtained results, and consequently, the use of AMFs provides a significant improvement in query performance, growing with the scale of the network.

7. Conclusions

In this research, we tried to find a way to make querying over a large number of datasources more feasible. By investigating this matter, large-scale decentralized social Web applications could become reality. The proposed method is to extend the client-side query engine by implementing source selection based on Approximate Membership Functions. By performing source selection, irrelevant datasources for a given input query can be filtered out, significantly reducing the number of HTTP requests to be executed by the client. Approximate Membership Functions can be given a false positive probability. A higher probability means a higher chance for an irrelevant source to end up in the selected list of sources. There is however a trade-off, as AMFs with a lower false positive probability are bigger in size.

Some experiments were set up, in order to measure the query performance of a query engine extended with AMFs (E'), in comparison to the default engine (E). The goal was to find that the difference in performance between E' and E increased with the expansion of the network of datasources. For this, three different metrics were measured: the number of HTTP requests, the total query execution time, and the client-side memory usage.

For all three metrics, we saw the same pattern. In very small networks, the metrics of E' and E are almost equal. However, the more datasources are added, the more the difference in performance between E' and E grows, with E' providing much better results than E. Some queries in the regular setup even failed due to timeouts or *heap out of memory* errors, while these queries caused no issues in the AMF setup.

Different false positive probabilities were also tested, in order to find the optimal balance between query performance, filter construction time and filter size. We discovered that a false positive rate of $p = 1/64$ proved to contribute to the best results overall.

Although the use of Approximate Membership Functions provides a significant improvement in query performance, the construction of these filters must also be considered. In a setup where the client itself must maintain the AMFs, the total number of HTTP requests invoked by a query is much higher than in a regular setup without AMFs. This causes the query response times to escalate significantly. However, two solutions have been proposed to this problem, both involving an external aggregator. In both solutions, this aggregator discovers new datasources in the network and keeps a list of all sources. Based on this list, it creates AMFs, either periodically or upon file changes.

The first solution requires the aggregator to combine the filters into one file and send it to the client when there is a query to be executed. The client must then perform source selection based on the filters in the file, after which the query can be executed. In the second solution, the aggregator must perform source selection itself, after it has received the query from the client. It then responds to the client with the selected set of sources.

Both of these methods add only one HTTP request to the complete querying process of the client, making the complete process still very performant in comparison to the regular query engine setup. However, the approaches differ in other ways. The first approach is not ideal for very large networks, as the combined file containing filters grows with the number of sources and must be sent over the network. Although the second approach does not have this issue, it requires more processing power at the aggregator to perform source selection. In addition, it is less secure regarding privacy, as the client’s query must be sent over the network and read by the aggregator.

To summarize, we can state that the use of Approximate Membership Functions when querying over a large number of sources provides a major improvement to query performance, but only if the resources are available to implement an external aggregator, which manages the AMFs. Unfortunately, even with the use of AMFs, the measured metrics are for some queries over many sources still rather poor to make responsive decentralized social Web applications possible. Nonetheless, the findings in this research might help us take a step closer to that goal.

8. Future research

Various subjects may be further investigated to continue the work of this research. In the discussion, we mentioned two solutions for reducing the number of HTTP requests using the proposed AMF-enabled query engine in combination with an aggregator. Although the optimal use cases and the advantages and disadvantages of each approach were briefly discussed, these methods must be further investigated and experimented with in future research, as the correct use of aggregators in a decentralized environment can significantly improve query performance.

Furthermore, other AMF approaches should be tested, such as compressed Bloom filters and Golomb-coded sets. These data structures are more space-efficient than regular Bloom filters. Therefore, they could be particularly useful in the first of the two mentioned solutions regarding aggregators, as the biggest issue there is that sending a large AMF file over the network drastically reduces query performance.

In fact, Approximate Membership Functions are not the only way of implementing source selection in a client-side query engine in the context of Solid. Other summarization approaches must be investigated and tested as well, such as those discussed in the literature study.

Finally, decentralized social Web environments could be simulated in larger scale, and queried over using more elaborate queries. We have a long way to go before we can achieve truly large-scale decentralized social Web applications, but the first steps have already been taken.

9. References

[1] T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web,” *Sci. Am.*, 2001.

[2] N. Shadbolt, W. Hall, and T. Berners-Lee, “The Semantic Web Revisited,” *IEEE Intell. Syst.*, vol. 21, no. 3, pp. 96–101, 2006, doi: 10.1109/MIS.2006.62.

[3] T. Berners-Lee, “Linked Data,” Jul. 27, 2006. https://www.w3.org/DesignIssues/LinkedData.html (accessed Oct. 15, 2020).

[4] J. P. McCrae *et al.*, “The Linked Open Data Cloud,” May 20, 2020. https://lod-cloud.net/ (accessed Mar. 23, 2021).

[5] A. V. Sambra *et al.*, “Solid: A Platform for Decentralized Social Applications Based on Linked Data,” 2016. Accessed: Oct. 08, 2020. [Online]. Available: https://diasporafoundation.org.

[6] R. Albert, H. Jeong, and A. L. Barabási, “Diameter of the world-wide web,” *Nature*, vol. 401, no. 6749, pp. 130–131, 1999, doi: 10.1038/43601.

[7] R. Cyganiak, D. Wood, and M. Lanthaler, “RDF 1.1 Concepts and Abstract Syntax,” Feb. 25, 2014. https://www.w3.org/TR/rdf11-concepts/ (accessed Oct. 21, 2020).

[8] C. Bizer, T. Heath, and T. Berners-Lee, “Linked data - The story so far,” *Int. J. Semant. Web Inf. Syst.*, vol. 5, no. 3, pp. 1–22, 2009, doi: 10.4018/jswis.2009081901.

[9] T. Berners-Lee, R. Fielding, and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” Jan. 2005. https://www.hjp.at/doc/rfc/rfc3986.html (accessed Oct. 22, 2020).

[10] C. Bizer, “SweoIG/TaskForces/CommunityProjects/LinkingOpenData - W3C Wiki,” 2010. https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData (accessed Oct. 25, 2020).

[11] S. Speicher, J. Arwe, and A. Malhotra, “Linked Data Platform 1.0,” Feb. 26, 2015. https://www.w3.org/TR/ldp/ (accessed Nov. 08, 2020).

[12] E. Miller, “An Introduction to the Resource Description Framework,” *Bull. Am. Soc. Inf. Sci. Technol.*, vol. 25, no. 1, pp. 15–19, Jan. 2005, doi: 10.1002/bult.105.

[13] S. Harris, A. Seaborne, and E. Prud’hommeaux, “SPARQL 1.1 Query Language,” Mar. 21, 2013. https://www.w3.org/TR/sparql11-query/ (accessed Oct. 24, 2020).

[14] J. Van Ossenbruggen, L. Hardman, and L. Rutledge, “Hypermedia and the Semantic Web: A Research Agenda,” 2001.

[15] O. Lassila, “Resource Description Framework (RDF) Model and Syntax Specification,” 1998. Accessed: Oct. 15, 2020. [Online]. Available: http://www.w3.org/1998/10/WD-rdf-syntax-19981008.

[16] G. Schreiber and Y. Raimond, “RDF 1.1 Primer,” Jun. 24, 2014. https://www.w3.org/TR/rdf11-primer/ (accessed Oct. 20, 2020).

[17] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan, “Extensible Markup Language (XML),” 2006. Accessed: Oct. 20, 2020. [Online]. Available: http://www.w3.org/TR/2006/REC-xml11-20060816.

[18] D. Berrueta and J. Phipps, “Best Practice Recipes for Publishing RDF Vocabularies,” 2008. https://www.w3.org/TR/swbp-vocab-pub/ (accessed Oct. 20, 2020).

[19] S. Bechhofer *et al.*, “OWL Web Ontology Language Reference,” Feb. 10, 2004. https://www.w3.org/TR/owl-ref/ (accessed Nov. 09, 2020).

[20] D. Brickley, R. V. Guha, and B. McBride, “RDF Schema 1.1,” Feb. 25, 2014. https://www.w3.org/TR/rdf-schema/ (accessed Nov. 09, 2020).

[21] R. Taelman, “Continuously Updating Queries over Real-Time Linked Data,” 2015.

[22] D. Beckett, T. Berners-Lee, E. Prud’hommeaux, and G. Carothers, “RDF 1.1 Turtle,” Feb. 25, 2014. https://www.w3.org/TR/turtle/ (accessed Oct. 20, 2020).

[23] B. Quilitz and U. Leser, “Querying distributed RDF data sources with SPARQL,” *Lect. Notes Comput. Sci.*, vol. 5021 LNCS, pp. 524–538, 2008, doi: 10.1007/978-3-540-68234-9_39.

[24] R. Buyle *et al.*, “Streamlining governmental processes by putting citizens in control of their personal data,” 2019.

[25] A. Sambra, A. Guy, S. Capadisli, and N. Greco, “Building Decentralized Applications for the Social Web,” in *Proceedings of the 25th International Conference Companion on World Wide Web - WWW ’16 Companion*, 2016, pp. 1033–1034, doi: 10.1145/2872518.2891060.

[26] T. Berners-Lee and R. Verborgh, “Solid: Linked Data for personal data management,” Oct. 08, 2018. https://rubenverborgh.github.io/Solid-DeSemWeb-2018/ (accessed Nov. 10, 2020).

[27] “Get a Pod · Solid.” https://solidproject.org/users/get-a-pod (accessed Nov. 10, 2020).

[28] R. Taelman, S. Steyskal, and S. Kirrane, “Towards Querying in Decentralized Environments with Privacy-Preserving Aggregation,” 2020.

[29] S. Capadisli and A. Guy, “Linked Data Notifications,” May 02, 2017. https://www.w3.org/TR/ldn/ (accessed Nov. 12, 2020).

[30] E. Prud’hommeaux and C. Buil-Aranda, “SPARQL 1.1 Federated Query,” Mar. 21, 2013. https://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/ (accessed Nov. 26, 2020).

[31] A. Sambra, H. Story, and T. Berners-Lee, “WebID 1.0,” Mar. 05, 2014. https://www.w3.org/2005/Incubator/webid/spec/identity/ (accessed Nov. 16, 2020).

[32] T. Inkster, H. Story, and B. Harbulot, “WebID-TLS,” Mar. 05, 2014.

https://www.w3.org/2005/Incubator/webid/spec/tls/ (accessed Nov. 17, 2020).

[33] "WebID-OIDC Authentication Spec." https://github.com/solid/webid-oidc-spec (accessed Dec. 14, 2020).

[34] P. Haase, T. Mathäß, and M. Ziller, "An evaluation of approaches to federated query processing over linked data," *ACM Int. Conf. Proceeding Ser.*, no. January, 2010, doi: 10.1145/1839707.1839713.

[35] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over Linked Data," *World Wide Web*, vol. 14, no. 5, pp. 495–544, 2011, doi: 10.1007/s11280-010-0107-z.

[36] H. Stuckenschmidt, R. Vdovjak, G. J. Houben, and J. Broekstra, "Index structures and algorithms for querying distributed RDF repositories," *Thirteen. Int. World Wide Web Conf. Proceedings, WWW2004*, pp. 631–639, 2004, doi: 10.1145/988672.988758.

[37] R. Verborgh, M. Vander, S. P. Colpaert, S. Coppens, E. Mannens, and R. Van De Walle, "Web-Scale Querying through Linked Data Fragments," *Proc. 7th Work. Linked Data Web*, vol. 1184, 2014, Accessed: Mar. 14, 2021. [Online]. Available: http://ceur-ws.org/Vol-1184/ldow2014_paper_04.pdf.

[38] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "FedX: Optimization techniques for federated query processing on linked data," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7031 LNCS, no. PART 1, pp. 601–616, 2011, doi: 10.1007/978-3-642-25073-6_38.

[39] M. Vander Sande, R. Verborgh, J. Van Herwegen, E. Mannens, and R. Van de Walle, "Opportunistic linked data querying through approximate membership metadata," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9366, no. Iswc, pp. 92–110, 2015, doi: 10.1007/978-3-319-25007-6_6.

[40] R. Taelman, J. van Herwegen, M. Vander Sande, and R. Verborgh, "Optimizing approximate membership metadata in triple pattern fragments for clients and servers," *CEUR Workshop Proc.*, vol. 2757, pp. 1–16, 2020.

[41] T. Neumann and G. Weikum, "Scalable Join Processing on Very Large RDF Graphs," *SIGMOD '09 Proc. 2009 ACM SIGMOD Int. Conf. Manag. data*, pp. 627–640, Jun. 2009, Accessed: Mar. 22, 2021. [Online]. Available: https://doi.org/10.1145/1559845.1559911.

[42] K. Hose and R. Schenkel, "Towards Benefit-Based RDF Source Selection for SPARQL Queries," *Swim '12 Proc. 4th Int. Work. Semant. Web Inf. Manag.*, pp. 1–8, May 2012, Accessed: Mar. 22, 2021. [Online]. Available: https://doi.org/10.1145/2237867.2237869.

[43] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970, Accessed: Mar. 22, 2021. [Online]. Available: https://doi.org/10.1145/362686.362692.

[44] F. Putze, P. Sanders, and J. Singler, "Cache-, Hash-, and space-efficient bloom filters," *ACM*

J. Exp. Algorithmics, vol. 14, Sep. 2008, doi: 10.1145/1498698.1594230.

[45] R. Taelman, “rubensworks/ldbc-snb-decentralized: A tool to create a decentralized version of the LDBC SNB social network dataset, and serve it over HTTP.,” 2021. https://github.com/rubensworks/ldbc-snb-decentralized (accessed Mar. 19, 2021).

[46] “The LDBC Social Network Benchmark.” Accessed: Mar. 19, 2021. [Online]. Available: https://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf.

[47] R. Verborgh, J. Van Herwegen, and R. Taelman, “solid/community-server: Community Solid Server: an open and modular implementation of the Solid specifications.” https://github.com/solid/community-server (accessed Mar. 20, 2021).

[48] R. Verborgh, R. Taelman, and M. Vander Sande, “LinkedDataFragments/Server.js: A Triple Pattern Fragments server for Node.js.” https://github.com/LinkedDataFragments/Server.js/tree/master (accessed Mar. 21, 2021).

[49] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, “Comunica: a Modular SPARQL Query Engine for the Web,” *Proc. 17th Int. Semant. Web Conf.*, vol. 11137, pp. 239–255, 2018.

[50] R. Taelman, “Comunica – SPARQL Architecture.” https://comunica.dev/docs/modify/advanced/architecture_sparql/ (accessed Mar. 21, 2021).

10. Appendices

Appendix A: SPARQL queries

The prefixes listed below are used in each of the 15 SPARQL queries executed during the experiments. As such, they will only be listed here.

```
PREFIX snvoc: <http://localhost:3000/www.ldbc.eu/ldbc_socialnet/
                1.0/vocabulary/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

Query 1

```
SELECT ?name WHERE {
    ?person snvoc:firstName ?name .
}
```

Query 2

```
SELECT ?person WHERE {
    ?person snvoc:firstName "Tom" .
}
```

Query 3

```
SELECT ?person WHERE {
    ?person snvoc:browserUsed "Firefox" .
}
```

Query 4

```
SELECT ?person WHERE {
    ?person snvoc:speaks "fr" .
}
```

Query 5

```
SELECT ?person WHERE {
    ?person snvoc:firstName "Tom" .
    ?person snvoc:gender "male" .
}
```

Query 6

```
SELECT ?person WHERE {
    ?person snvoc:speaks "nl" .
    ?person snvoc:gender "female" .
}
```

Query 7

```
SELECT ?person WHERE {
    ?person snvoc:speaks "nl" .
    ?person snvoc:browserUsed "Internet Explorer" .
}
```

Query 8

```
SELECT ?name WHERE {
  ?person snvoc:id "4398046512167"^^<http://www.w3.org/2001/XMLSchema#long> .
  ?person snvoc:firstName ?name .
}
```

Query 9

```
SELECT ?friend WHERE {
  ?person snvoc:id "4398046512167"^^<http://www.w3.org/2001/XMLSchema#long> .
  ?person snvoc:knows ?friend .
}
```

Query 10

```
SELECT ?friend WHERE {
    ?person snvoc:firstName "Tom" .
    ?person snvoc:knows ?friend .
}
```

Query 11

```
SELECT ?person WHERE {
    ?person snvoc:firstName "Tom" .
    ?person snvoc:gender "male" .
    ?person snvoc:speaks "en" .
}
```

Query 12

```
SELECT ?person WHERE {
    ?person snvoc:speaks "nl" .
    ?person snvoc:hasInterest
<http://localhost:3000/www.ldbc.eu/ldbc_socialnet/1.0/tag/J._R._R._Tolkien> .
}
```

Query 13

```
SELECT ?city WHERE {
  ?person snvoc:id "4398046512167"^^<http://www.w3.org/2001/XMLSchema#long> .
  ?person snvoc:isLocatedIn ?place .
  ?place foaf:name ?city .
}
```

Query 14

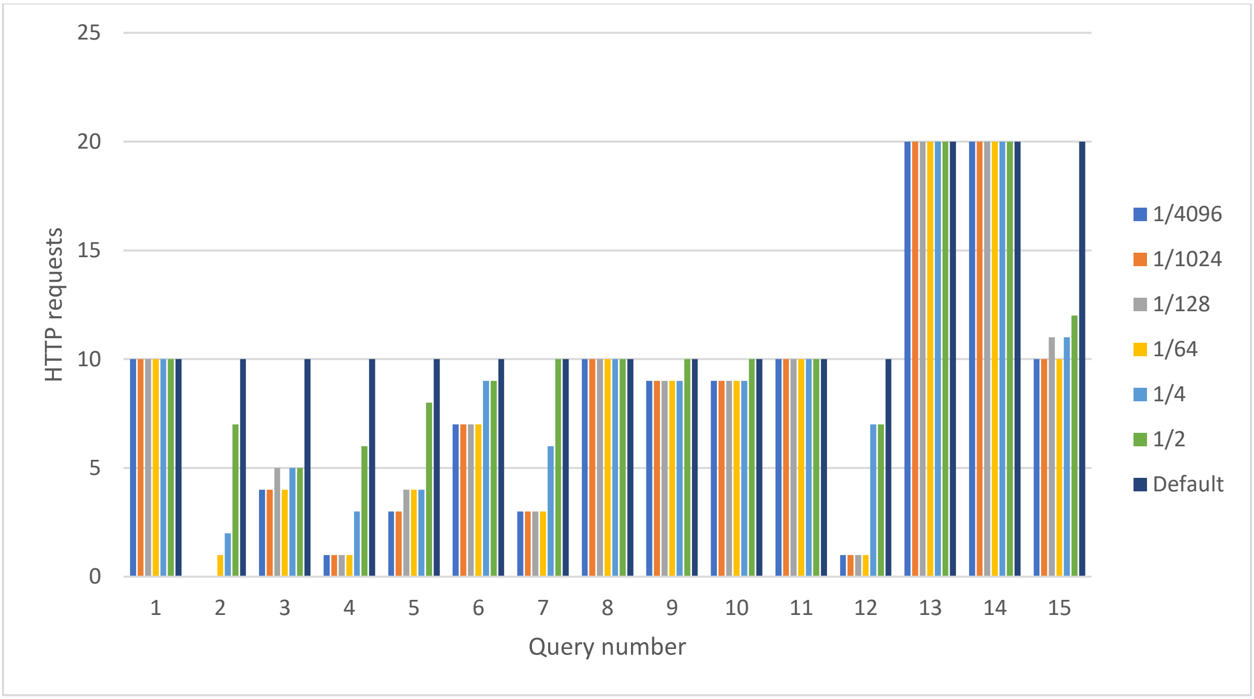
```
SELECT ?city WHERE {
    ?person snvoc:firstName "Tom" .
    ?person snvoc:isLocatedIn ?place .
    ?place foaf:name ?city .
}
```

Query 15

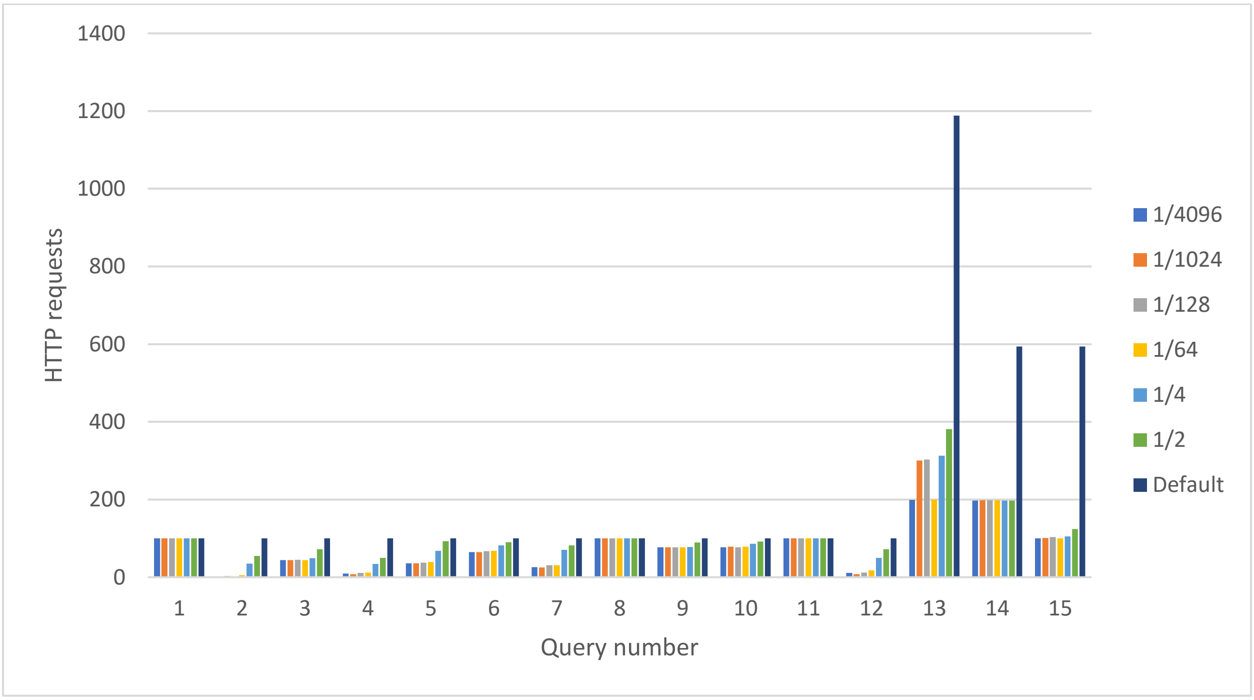
```
SELECT ?name WHERE {
    ?place foaf:name "Brussels" .
    ?person snvoc:isLocatedIn ?place .
    ?person snvoc:firstName ?name .
}
```

Appendix B: Number of HTTP requests per query

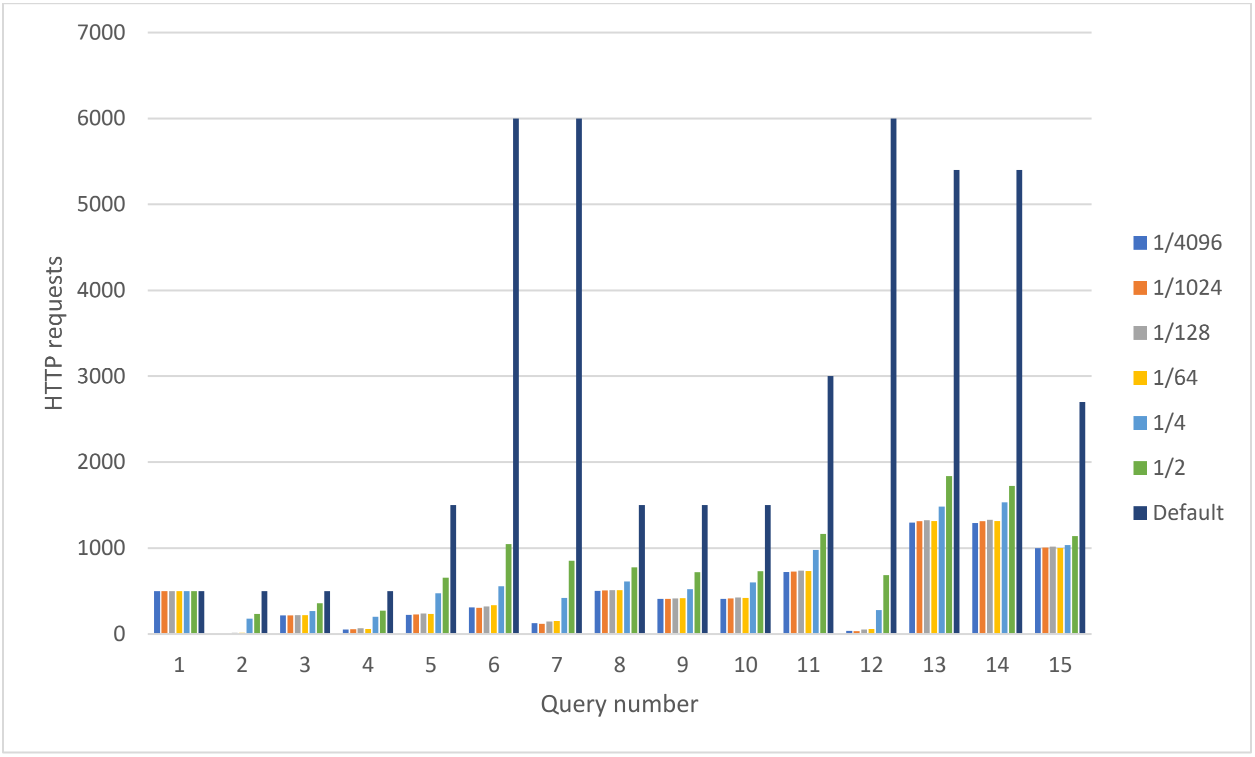
10 datasources



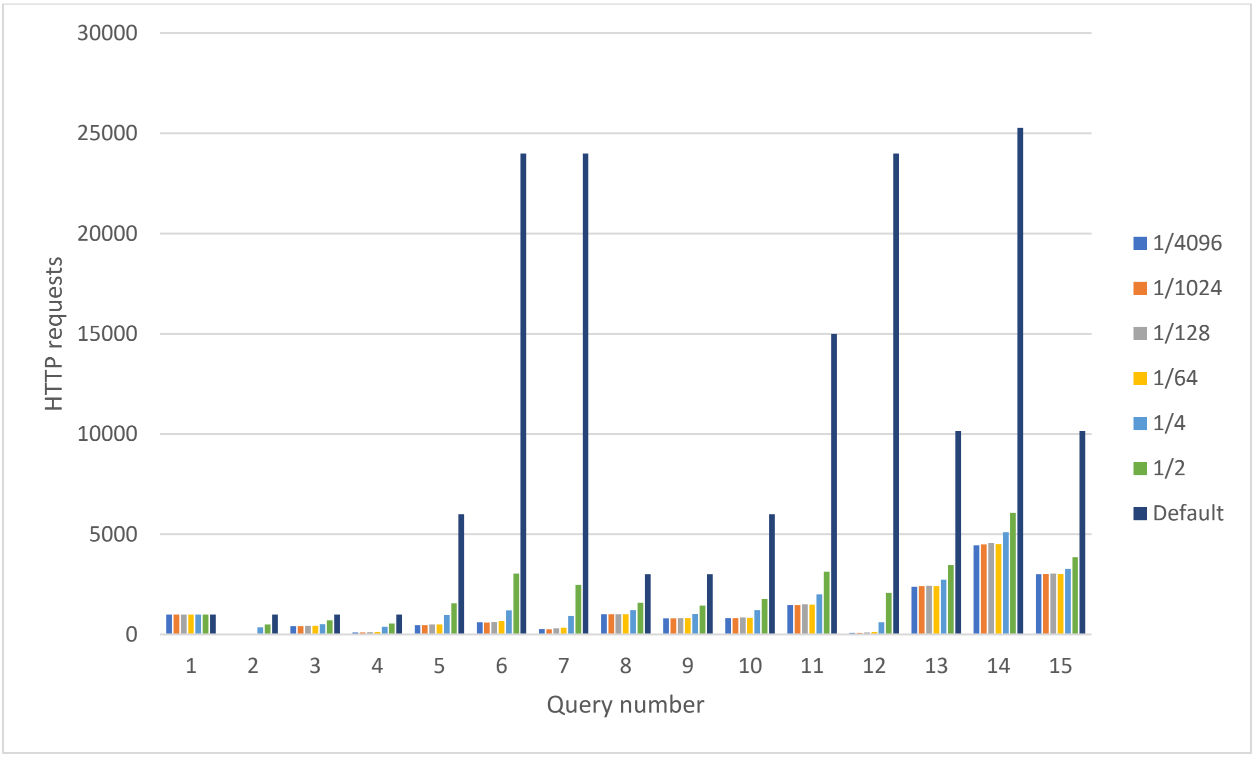
100 datasources



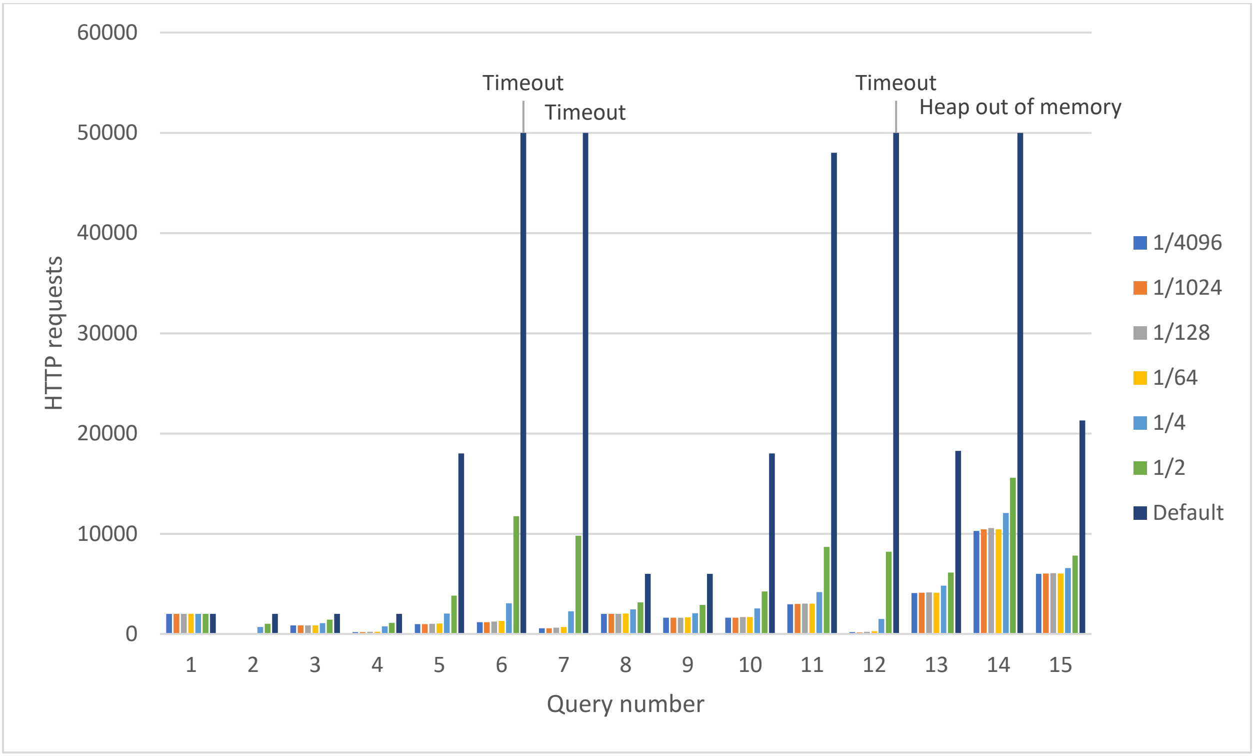
500 datasources



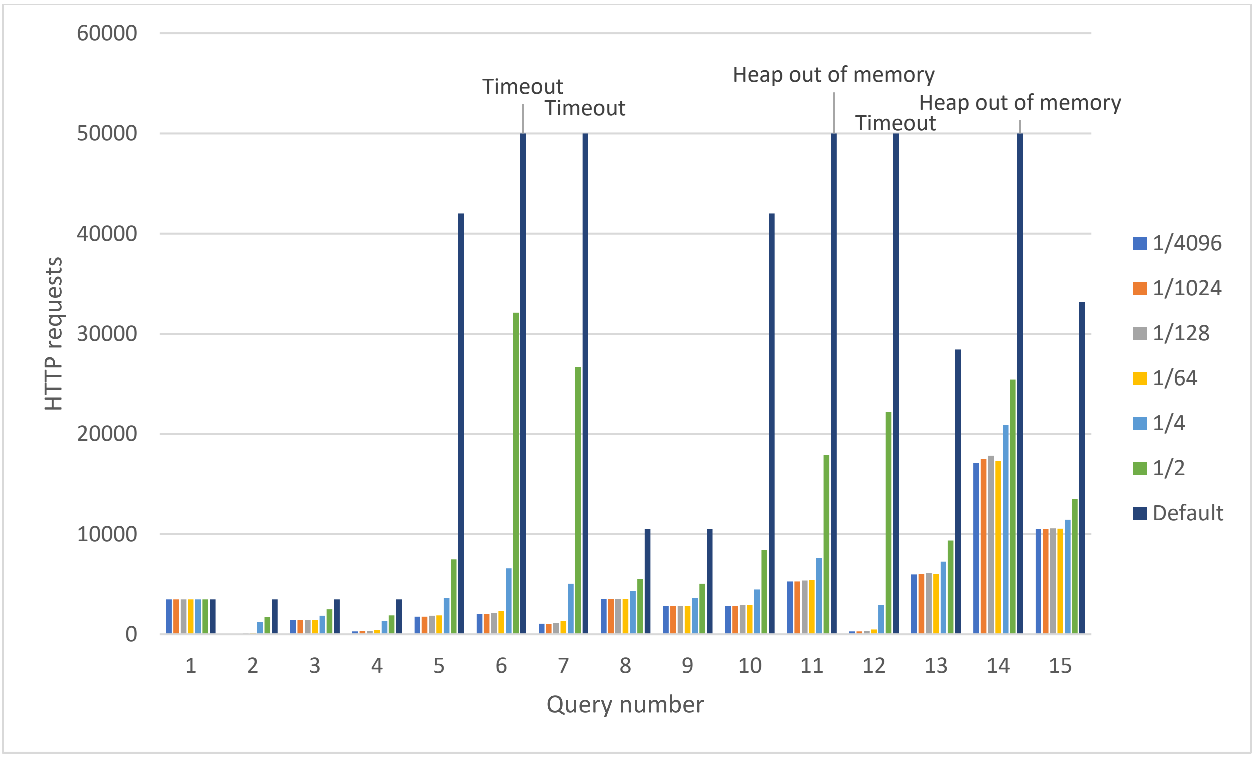
1000 datasources



2000 datasources

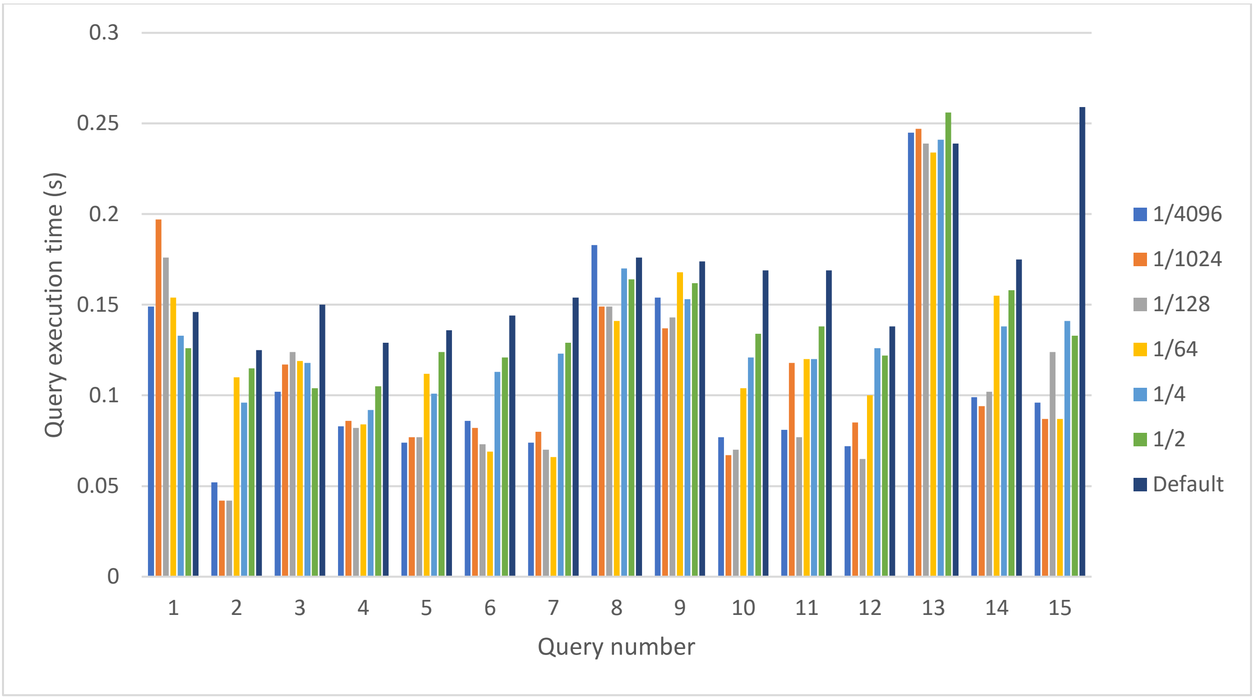


3500 datasources

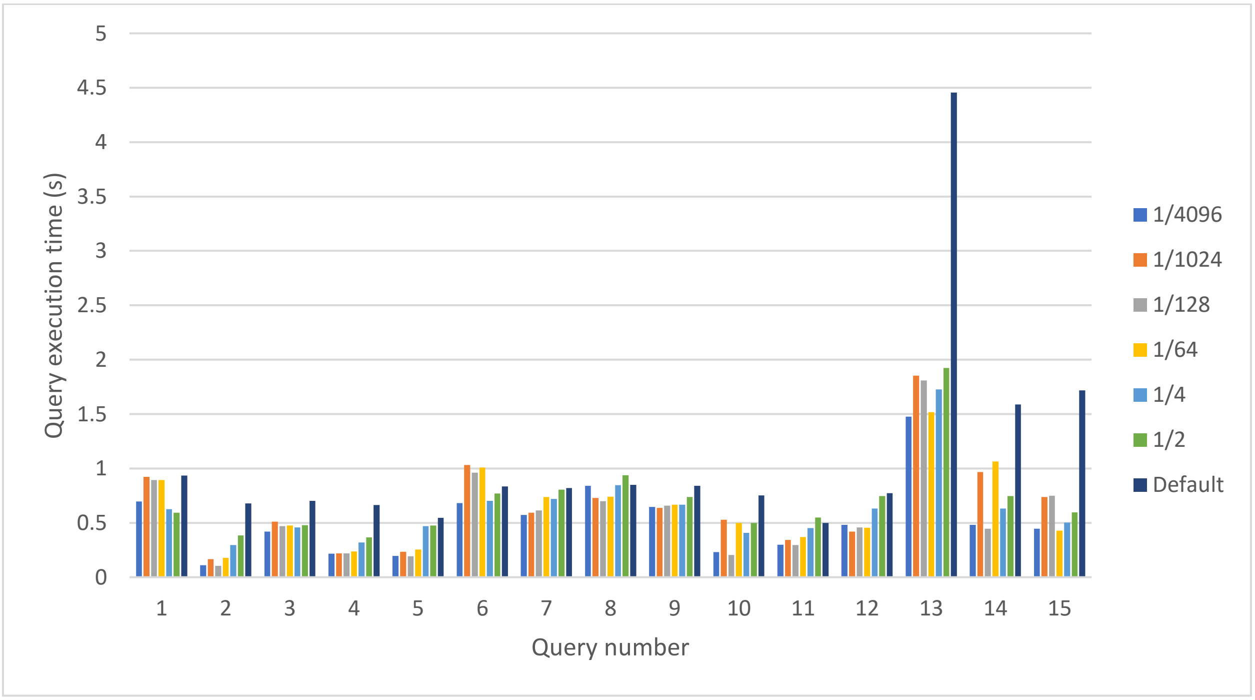


Appendix C: Query response time per query

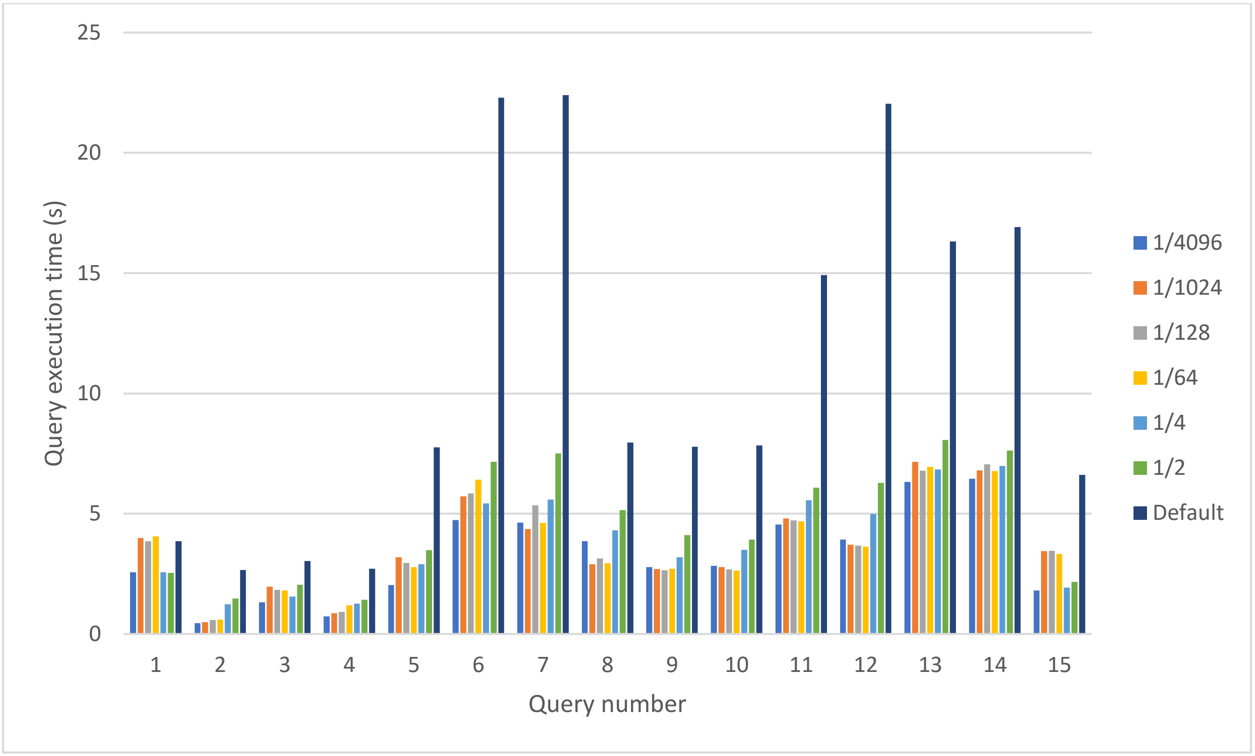
10 datasources



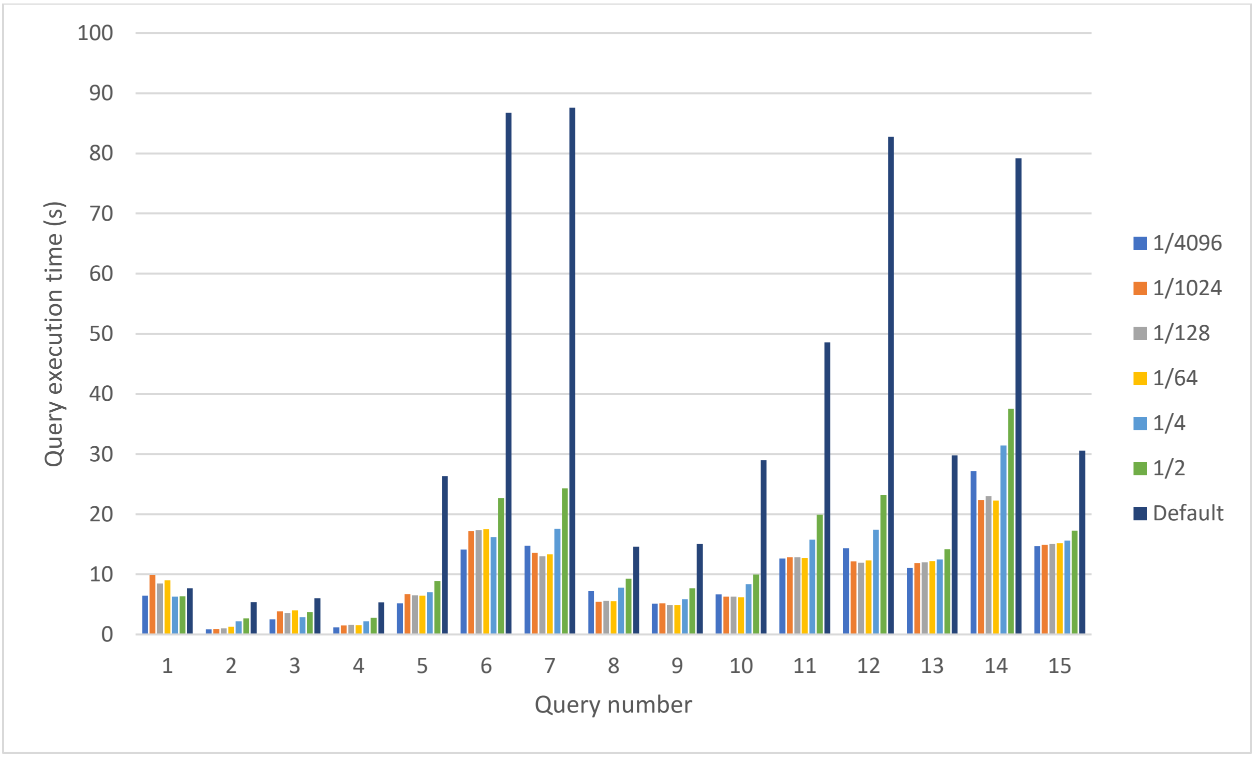
100 datasources



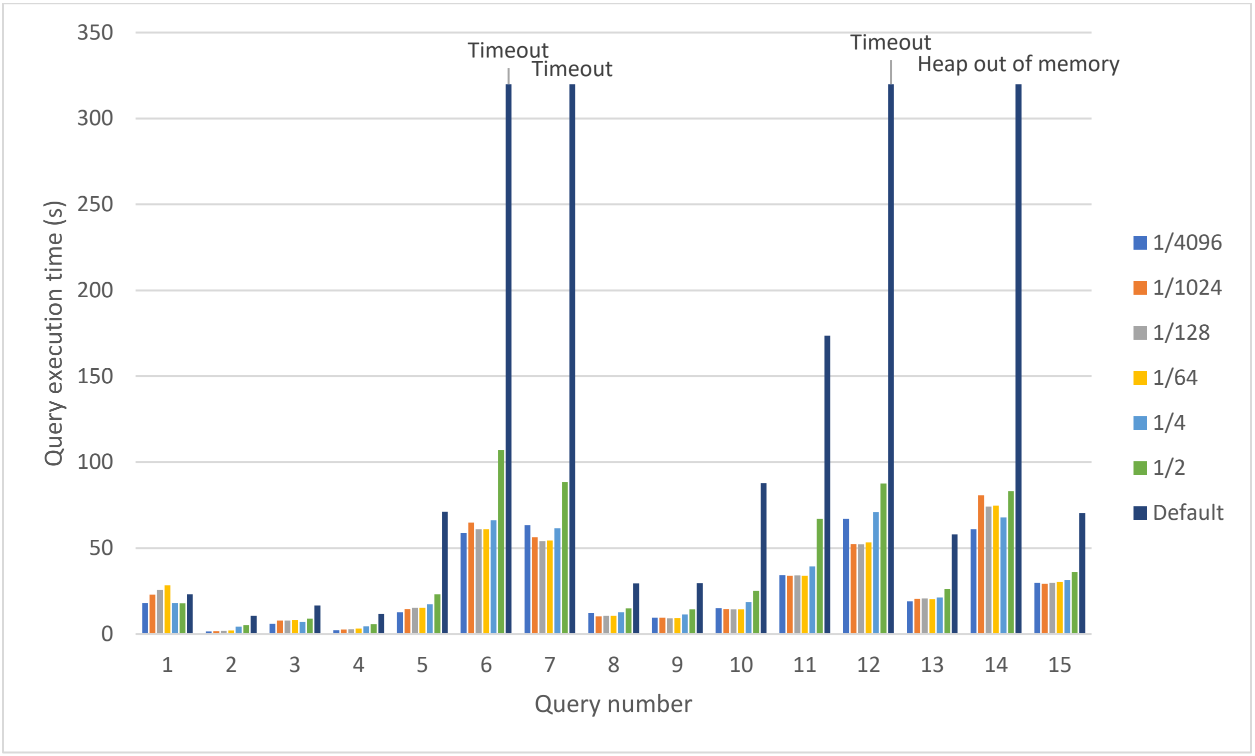
500 datasources



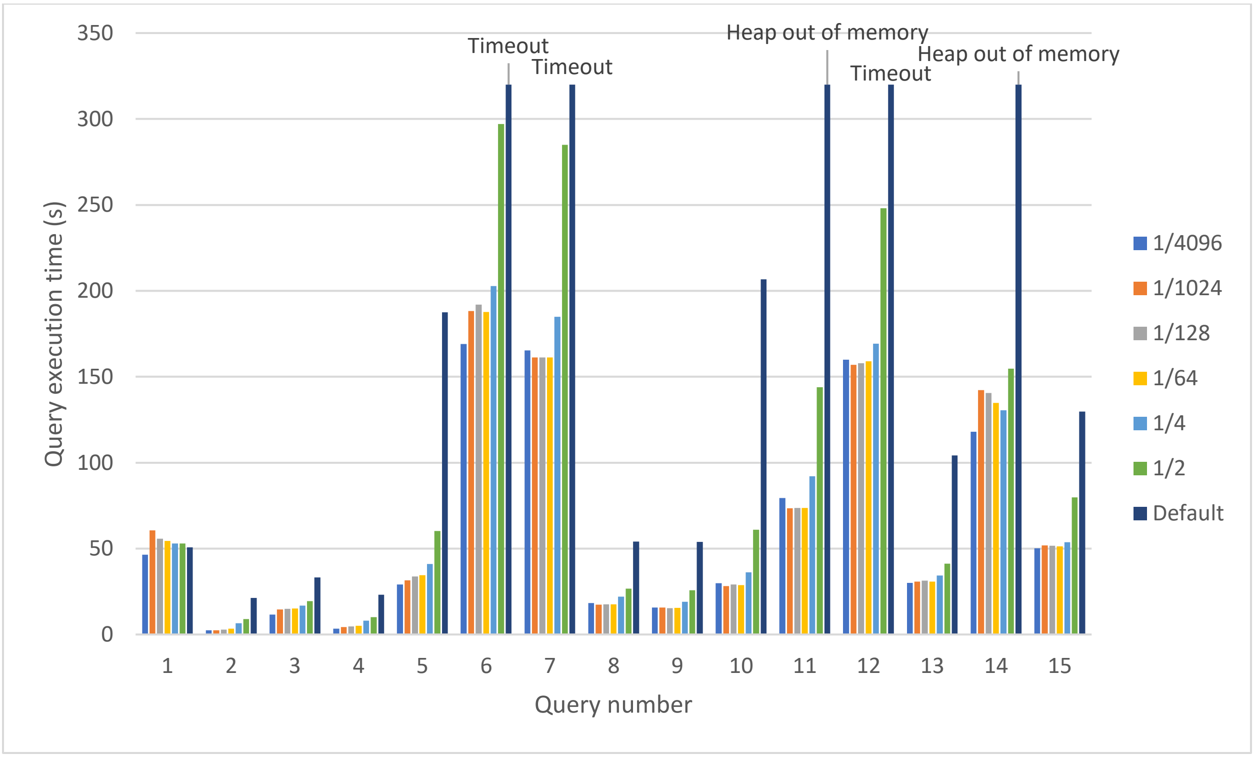
1000 datasources



2000 datasources

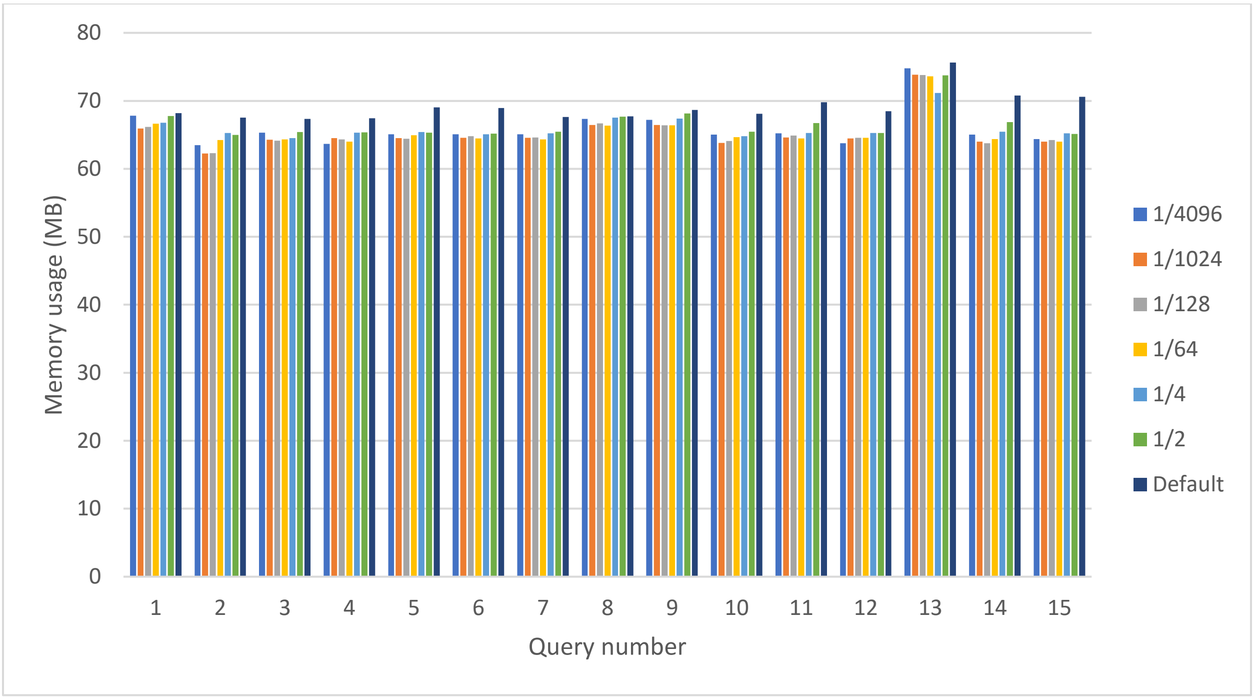


3500 datasources

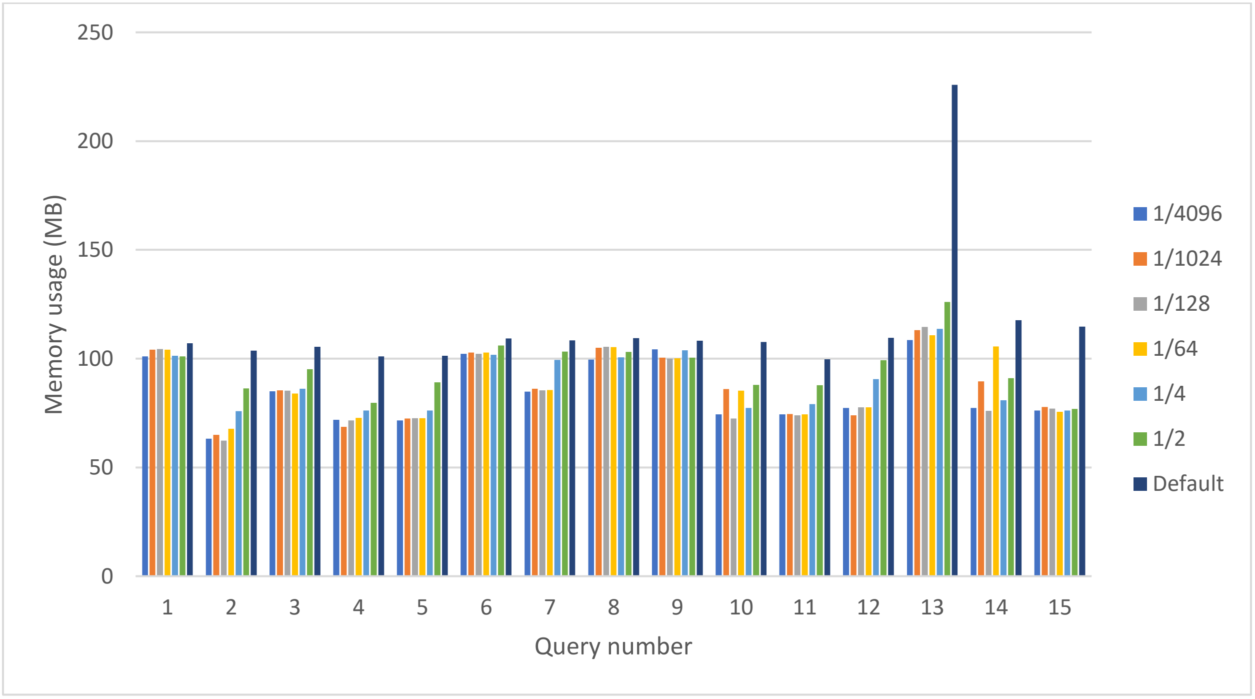


Appendix D: Memory usage per query

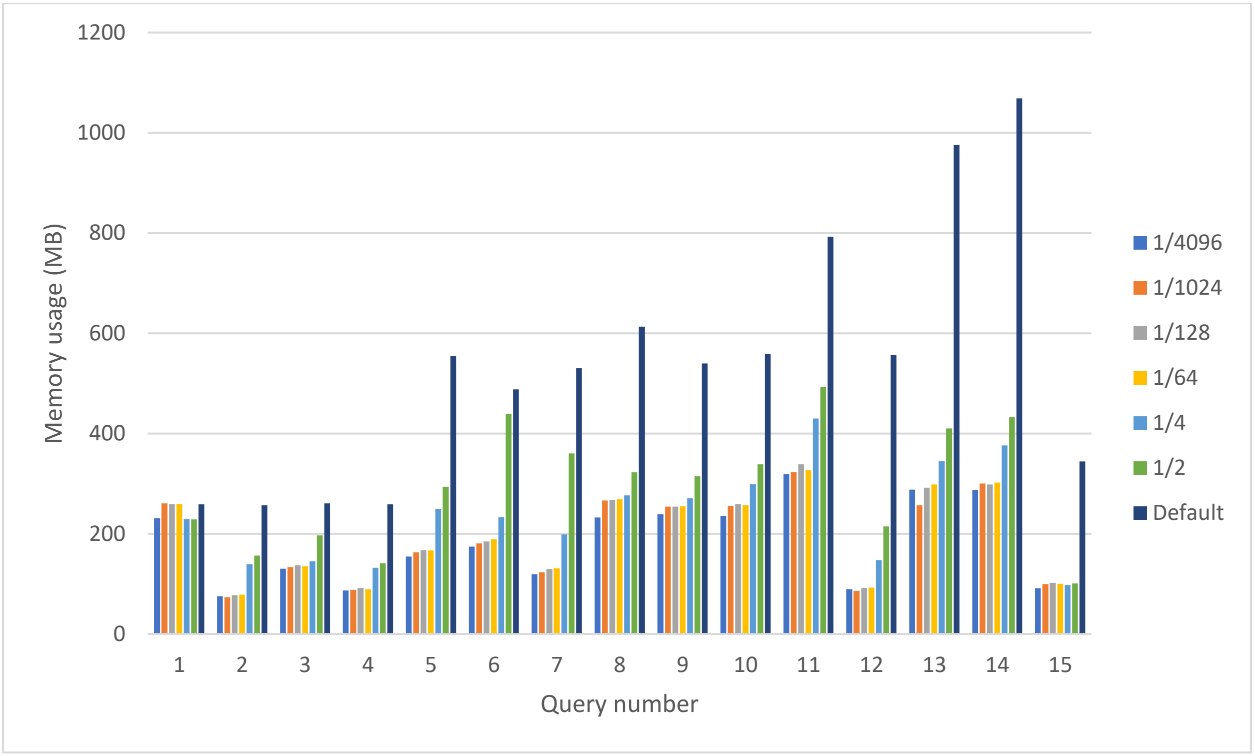
10 datasources



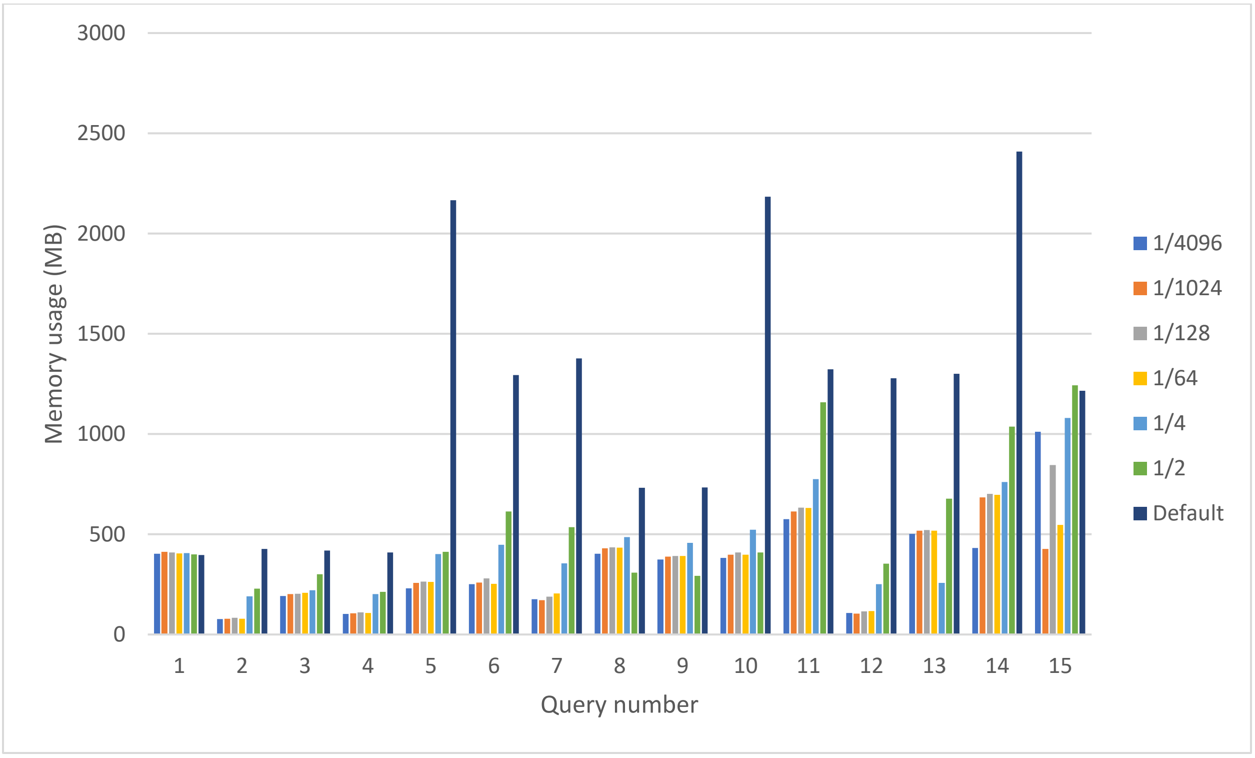
100 datasources



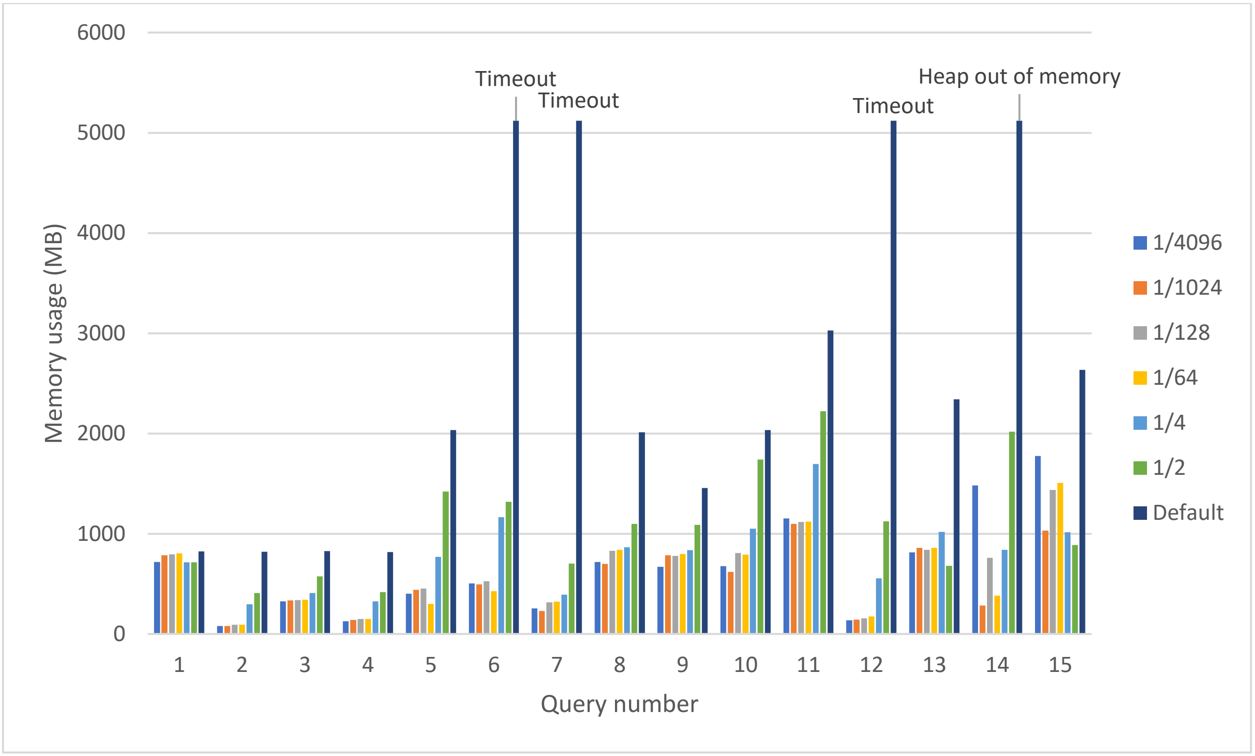
500 datasources



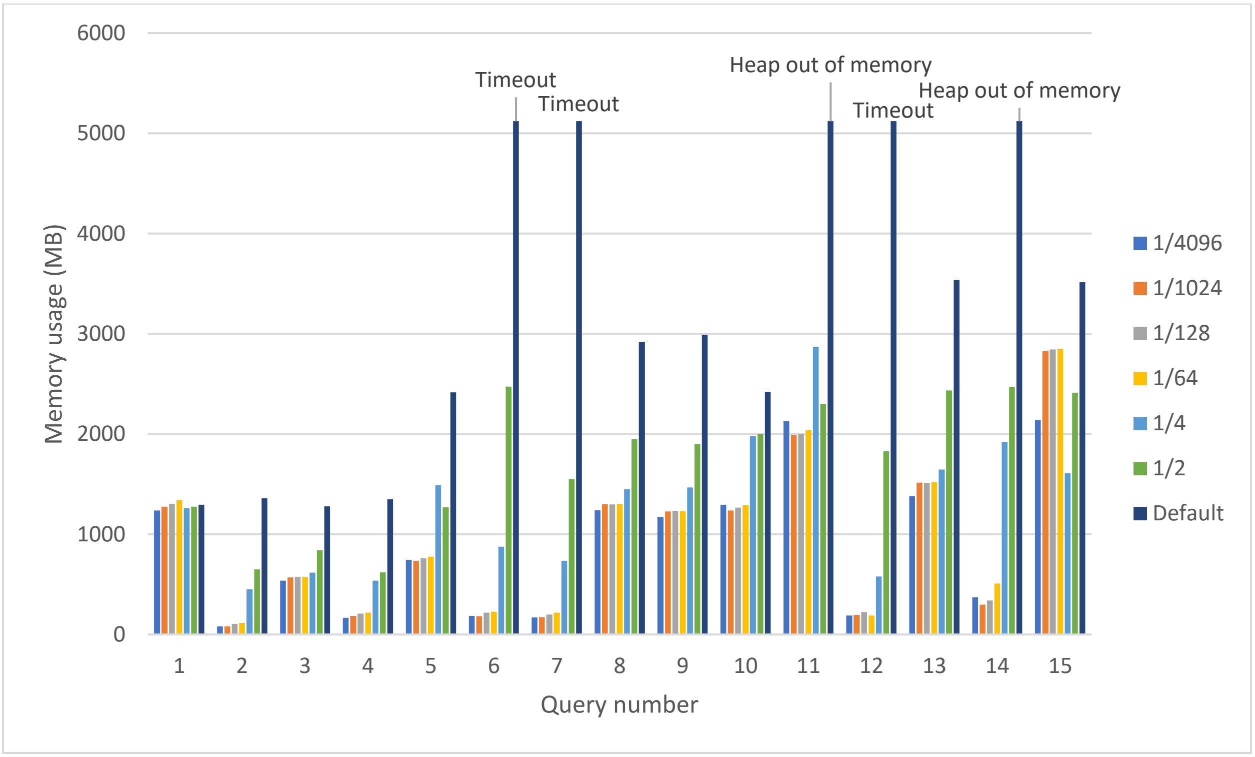
1000 datasources



2000 datasources



3500 datasources



FACULTY OF ENGINEERING
AND ARCHITECTURE

Scaling networks of personal data stores through Approximate Membership Functions

Thomas Devriese
Student number: 01806904

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. ir. Ruben Verborgh

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2020-2021

GHENT
UNIVERSITY