Peer-to-peer Search over Linked Data in the Research and Education Space

Eindhoven, 13th May 2015

Author: Simeon van der Steen
simeon@simeonvandersteen.nl

Supervisor and tutor: dr. G.H.L. Fletcher
g.h.l.fletcher@tue.nl

External supervisor: M. McRoberts
mo.mcroberts@bbc.co.uk

Assessment committee: dr. G.H.L. Fletcher
dr. A. Serebrenik
dr. M.A. Westenberg
Abstract

Many public services maintain their own archives, with data stored for one to be able to access it any time in the future. However, access to the material in those archives is hard in practice. The Research and Education Space (RES) is a partnership between the BBC, Jisc, and the BUFVC, which tries to address this problem by opening up archives from several services, and making their catalogue data searchable. In this thesis we present a Linked Data query engine that can be used as a foundation for a scalable and fault-tolerant search API for RES. Fault-tolerance is important because if a part of the collections in the system is unavailable, the other collections should still be available for querying. We define the structure of a query as a keyword or a predicate/object pair of an RDF triple. These query parts can be combined into a conjunctive query. To process queries we design a query engine: a peer-to-peer network that consists of a routing application routing queries to relevant nodes in the network, and a database application that performs the actual search. Both applications run on every node in the network, so that they all act as individual APIs for the system as a whole, and making the system fault-tolerant. The system has low requirements, both applications run on any common Linux-based server environment. By testing the performance of the system we show that it is capable of meeting the requirements for RES and that it can be used for beta testing in a production environment. However, in its current implementation some improvements are possible. We conclude the thesis with a discussion of these possibilities and other future work directions towards full deployment.
Preface

This thesis is the written result of my master’s project for the master Computer Science & Engineering at the Eindhoven University of Technology. The work was done during my job at the British Broadcasting Corporation in London, from November 2013 until May 2015. I feel the need to elaborate some more on this period than perhaps is usual, but it has been an extraordinary time for me, and those around to me.

After an already hectic start of the project in 2013, I received the news in March 2014 that I had to interrupt my project and go back to the Netherlands for serious medical treatment, just at the time I was planning on when to finish the project and complete my master’s. For the next nine months I was busy visiting hospitals, undergoing treatments and tests, and waiting for what was coming next, without being able to do any work. In this period I’ve had invaluable support and understanding from the Archive Development team at the BBC, from the Department of Mathematics and Computer Science at the TU/e, and my friends and family, with whom I got to share this extraordinary time. For all of that, I am very grateful. From January 2015 I was able to continue my project, in London, and now I am about to finish it and obtain my master’s degree. This I could only dream about one year ago, and would probably still be dreaming about if it were not for the people around me.

For the support I received in doing this project, I would like to thank a few people in particular (and in no specific order). I would like to thank Leo Simons and Dirk-Willem van Gulik for helping me getting in touch with the Archive Development team, and Catherine Morrissey and Jake Berger for their effort to get me on the team. I also want to thank Jake for showing me London’s best kept secrets, which you will never find as someone from abroad. I’d like to thank Alex Tucker, Andy Dudfield and Chiara del Vescovo for pointing out many typos, spelling and grammatical errors, and suggestions on how to improve my work. I’m grateful to George Fletcher for his guidance, advice and feedback on my work during the entire project, and to Mo McRoberts for his endless patience in discussing many aspects of the project, and his feedback on my work. Finally, of course, I would like to thank my friends and family, for their support during this time in my life.

I’ve had a wonderful experience and an inspiring time working for this team. They gave me the opportunity to not only start working on this project, but also to continue and complete it after nine months of absence. For this I am truly grateful.

Simeon van der Steen
Contents

1 Introduction ................................. 1
   1.1 Background ................................ 1
   1.2 Problem statement ......................... 2
   1.3 Contributions ............................. 2
   1.4 Thesis outline ............................. 3

2 Preliminaries ............................... 4
   2.1 Linked Data ............................... 4
      2.1.1 Motivation ............................ 4
      2.1.2 Linked Data ........................... 4
      2.1.3 Technologies used in Linked Data ....... 5
      2.1.4 Publishing Linked Data ................. 7
   2.2 Querying Linked Data ..................... 7
      2.2.1 Traditional approaches of querying distributed data .. 8
      2.2.2 Querying multiple Linked Data resources ... 8
      2.2.3 Data warehousing ....................... 9
      2.2.4 Federated query processing ............... 9
      2.2.5 Active discovery based query federation ... 10
      2.2.6 Link traversal based query execution .... 10
      2.2.7 Peer-to-peer query execution ............ 10
   2.3 Peer-to-peer querying ..................... 10
      2.3.1 Traditional peer-to-peer systems ....... 11
      2.3.2 Modern peer-to-peer networks .......... 11

3 Design ...................................... 12
   3.1 Requirements and assumptions ............. 12
   3.2 High-level design .......................... 13
      3.2.1 Centralised ............................ 13
      3.2.2 Decentralised ......................... 13
      3.2.3 Peer-to-peer ......................... 14
   3.3 Design of the network .................... 15
      3.3.1 A query ................................ 15
      3.3.2 A structured network ................. 15
      3.3.3 Storing routing information .......... 16
      3.3.4 Using routing information .......... 16
      3.3.5 Distributed hash table settings ....... 16
   3.4 Design of a single node ................. 19
      3.4.1 The database .......................... 19
      3.4.2 Ingesting the data resources .......... 21
   3.5 Expected performance and bottlenecks .... 21

4 Implementation ............................ 22

Peer-to-peer Search over Linked Data in the Research and Education Space
## CONTENTS

4.1 The database application ........................................... 22
  4.1.1 Full-text search with LuceneClient .......................... 22
  4.1.2 Predicate/object search with BerkeleyDBClient .............. 23
  4.1.3 Query result construction with Intersect and DBInterface .. 23
  4.1.4 Query processing with DBClient ............................. 24
  4.1.5 Accepting connections with Server .......................... 25
  4.1.6 Handling routed requests with RequestHandler .............. 25

4.2 The DHT and routing application ................................. 26
  4.2.1 Riak as the DHT .............................................. 26
  4.2.2 Accepting client connections with ThreadedTCPServer ..... 27
  4.2.3 Handling client requests with RequestHandler .............. 27
  4.2.4 Updating Riak ................................................ 27
  4.2.5 Retrieving and storing routing information with RiakClient .. 28

4.3 Communication between nodes ..................................... 29

4.4 Ingesting the data resources .................................... 30
  4.4.1 Parsing ....................................................... 30
  4.4.2 Indexing predicate/object pairs into Berkeley DB .......... 30
  4.4.3 Indexing literal objects into Lucene ......................... 30

5 Performance Testing Setup ......................................... 32
  5.1 Data selection ................................................. 32
  5.2 Single node tests ............................................. 33
    5.2.1 Parameters ............................................... 33
    5.2.2 Data preparation ........................................ 33
    5.2.3 Single node query execution test ........................ 34
    5.2.4 Single node load test ................................... 34
  5.3 Network tests ................................................ 35
    5.3.1 Parameters ............................................... 35
    5.3.2 Data preparation ........................................ 35
    5.3.3 Network query execution test ............................ 36
    5.3.4 Network load test ....................................... 36

6 Test Results ........................................................ 38
  6.1 Single node tests ............................................. 38
    6.1.1 Single node query execution test ......................... 38
    6.1.2 Single node load test ................................... 41
  6.2 Network tests ................................................ 43
    6.2.1 Network query execution test ............................ 43
    6.2.2 Network load test ....................................... 44
  6.3 Summary of the test results .................................. 47

7 Conclusions .......................................................... 48
  7.1 Contributions ................................................. 48
  7.2 Limitations .................................................... 49
  7.3 Future work .................................................... 49
    7.3.1 Production ready .......................................... 49
    7.3.2 Research ................................................ 50

A Software and hardware dependencies ............................. 51
  A.1 Database application .......................................... 51
  A.2 Routing application ......................................... 51
  A.3 Testing ......................................................... 52
    A.3.1 JMeter ..................................................... 52
    A.3.2 OpenNebula ................................................ 52
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of the traditional World Wide Web</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of the Semantic Web</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>The three parts of an RDF triple: subject, predicate (or property) and object</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>An RDF triple depicting the relation between Lost in Translation and Bill Murray</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>A decision tree of the considered approaches</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Query routing and result retrieval in the peer-to-peer network</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>The query execution process in the database application</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>A class diagram of DbInterface and its implementations</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>A class diagram of the most important classes in the database application</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>A class diagram of the most important classes in the routing application</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>The protocol used in communication between nodes</td>
<td>29</td>
</tr>
<tr>
<td>6.1</td>
<td>Execution time of queries from the British Museum Collections Online</td>
<td>39</td>
</tr>
<tr>
<td>6.2</td>
<td>Execution time of queries from the British Museum Collections Online – collection size of 80 million RDF triples only</td>
<td>40</td>
</tr>
<tr>
<td>6.3</td>
<td>Execution time of query 12 from the British Museum Collections Online</td>
<td>40</td>
</tr>
<tr>
<td>6.4</td>
<td>Execution time of queries 4, 5 and 6 from the British Museum Collections Online</td>
<td>40</td>
</tr>
<tr>
<td>6.5</td>
<td>Execution time of queries from the British National Bibliography collection</td>
<td>41</td>
</tr>
<tr>
<td>6.6</td>
<td>Single node load test – results for a target throughput</td>
<td>42</td>
</tr>
<tr>
<td>6.7</td>
<td>Single node load test – average ping time for a target throughput</td>
<td>42</td>
</tr>
<tr>
<td>6.8</td>
<td>Network query execution test</td>
<td>44</td>
</tr>
<tr>
<td>6.9</td>
<td>Network load test – average query time for a target throughput of 100 queries per second in a network of 20 nodes</td>
<td>45</td>
</tr>
<tr>
<td>6.10</td>
<td>Network load test – average ping time for a target throughput of 100 queries per second in a network of 20 nodes</td>
<td>46</td>
</tr>
<tr>
<td>6.11</td>
<td>Network load test – average network load for a target throughput of 100 queries per second in a network of 20 nodes</td>
<td>46</td>
</tr>
<tr>
<td>C.1</td>
<td>Query 1</td>
<td>57</td>
</tr>
<tr>
<td>C.2</td>
<td>Query 2</td>
<td>57</td>
</tr>
<tr>
<td>C.3</td>
<td>Query 3</td>
<td>58</td>
</tr>
<tr>
<td>C.4</td>
<td>Query 4</td>
<td>58</td>
</tr>
<tr>
<td>C.5</td>
<td>Query 5</td>
<td>58</td>
</tr>
<tr>
<td>C.6</td>
<td>Query 6</td>
<td>58</td>
</tr>
<tr>
<td>C.7</td>
<td>Query 7</td>
<td>58</td>
</tr>
<tr>
<td>C.8</td>
<td>Query 8</td>
<td>58</td>
</tr>
<tr>
<td>C.9</td>
<td>Query 9</td>
<td>58</td>
</tr>
<tr>
<td>C.10</td>
<td>Query 10</td>
<td>59</td>
</tr>
<tr>
<td>C.11</td>
<td>Query 11</td>
<td>59</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>C.12</td>
<td>Query 12</td>
<td>59</td>
</tr>
<tr>
<td>C.13</td>
<td>Query 1</td>
<td>60</td>
</tr>
<tr>
<td>C.14</td>
<td>Query 2</td>
<td>60</td>
</tr>
<tr>
<td>C.15</td>
<td>Query 3</td>
<td>60</td>
</tr>
<tr>
<td>C.16</td>
<td>Query 4</td>
<td>60</td>
</tr>
<tr>
<td>C.17</td>
<td>Query 5</td>
<td>60</td>
</tr>
<tr>
<td>C.18</td>
<td>Query 6</td>
<td>60</td>
</tr>
<tr>
<td>C.19</td>
<td>Query 7</td>
<td>60</td>
</tr>
<tr>
<td>C.20</td>
<td>Query 8</td>
<td>61</td>
</tr>
<tr>
<td>C.21</td>
<td>Query 9</td>
<td>61</td>
</tr>
<tr>
<td>C.22</td>
<td>Query 10</td>
<td>61</td>
</tr>
<tr>
<td>C.23</td>
<td>Query 11</td>
<td>61</td>
</tr>
<tr>
<td>C.24</td>
<td>Query 12</td>
<td>61</td>
</tr>
<tr>
<td>C.25</td>
<td>Average ping time for a target throughput of 100 queries per second</td>
<td>62</td>
</tr>
<tr>
<td>C.26</td>
<td>Average ping time for a target throughput of 200 queries per second</td>
<td>62</td>
</tr>
<tr>
<td>C.27</td>
<td>Average ping time for a target throughput of 300 queries per second</td>
<td>62</td>
</tr>
<tr>
<td>C.28</td>
<td>Average ping time for a target throughput of 400 queries per second</td>
<td>62</td>
</tr>
<tr>
<td>C.29</td>
<td>Average ping time for a target throughput of 500 queries per second</td>
<td>63</td>
</tr>
<tr>
<td>C.30</td>
<td>Average ping time for a target throughput of 600 queries per second</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Finding relevant nodes ................................................. 17
2. Performing a merge-join of two indexes ......................... 20
Chapter 1

Introduction

This document is a master’s thesis of the Web Engineering research group, in the Mathematics and Computer Science department of the Eindhoven University of Technology. It describes a project that has been conducted for the British Broadcasting Corporation (BBC) during 2014 and early 2015.

This chapter gives an introduction into the project. The motivation for the project is described in the next section, the main problem to be investigated and solved in this project in Section 1.2, and an outline of the thesis in Section 1.4.

1.1 Background

Many public services maintain their own archives, with data stored for one to be able to access it any time in the future. These archives usually are large, containing so much information spanning several centuries, that it can be considered cultural heritage. This immense source of information can be of great value to the general public, and since these archives are maintained by public services, they are maintained on behalf of the tax-payer, and should thus be accessible to the general public.

However, access to the material in those archives is hard in practice. The data is distributed over various physical locations (libraries, universities, broadcasters), in various – often analogue – formats.

The BBC – having one of the largest broadcast archives in the world\footnote{http://www.bbc.co.uk/blogs/legacy/aboutthebbc/2011/10/the-new-bbc-archive-centre-at.shtml} – decided to set the example by developing new ways to make use of their archive. A new department by the name of Archive Development\footnote{http://www.bbc.co.uk/pressoffice/pressreleases/stories/2008/10_october/10/ageh.shtml} initiated a vision called the Digital Public Space (DPS)\footnote{http://cl.ly/2Q1D2i3O021K/download/DPS.pdf}. Its goal is to have public services – or collection holders – open up their archives for the general public, and publish their catalogue data as Linked Open Data – machine-readable data under an open license \cite{33}. It is therefore not just about the BBC archives: the goal is creating a new standard of accessing publicly accessible data.

The Research and Education Space (RES)\footnote{http://bbc.in/res} is a partnership project between the BBC, Jisc\footnote{http://jisc.ac.uk/} and the British Universities Film & Video Council (BUFVC)\footnote{http://bufvc.ac.uk/}, and is the first step towards the DPS. Its deliverable is a platform that gives all levels of education and research access to archive catalogue data – and actual content – from for example the National Archives, Science Museum...
Collections, and British Film Institute Archive Collections. The material will be made available under a license suitable for use in education and research, and simplifying the legal aspects of the DPS. The platform will provide an API upon which applications can be built, to be used in classrooms and research.

1.2 Problem statement

The current version of the platform is designed and implemented as a centralised index, which ingests collections of Linked Data resources provided by collection holders, and makes them discoverable via an API. The centralised design of the platform is a potential problem, for two reasons.

First, when an increasing number of collection holders join the project, and therefore a very large amount of data will be ingested, the platform will hit its scalability limits. The second reason is that collection holders should be allowed to join and leave the project without too much hassle. They should preferably be able to join the project with their own resources (in terms of server capacity), so that in the end the project will be – at least partly – funded by the collection holders, instead of only the initiators.

For this project, we summarised the description above into the following problem statement:

Design and implement a Linked Data query engine that can be used as a foundation for querying decentralised Linked Data collections in the RES platform.

The problem can be divided into the following subproblems:

P1 Design of a formal query language

What is the structure of a query and what does it mean to have a result for a query?

P2 Design and implementation of the query engine

The main application to be designed and implemented, using the best suitable internal data structures.

P3 Test the performance of the engine

What is the performance of the query engine, will it be sufficient for the RES platform, and what are the bottlenecks?

In the remainder of this chapter we summarise the main contributions of this thesis, in Section 1.3, and give an outline of the rest of the thesis, in Section 1.4.

1.3 Contributions

This project contributes a peer-to-peer system in which multiple collections of Linked Data can be queried. Each peer runs an autonomous database application, and an application to route queries through the network. The system is fault-tolerant in a sense that an unavailable peer will not break the system as a whole. A routing strategy is used to prevent the network from flooding.

This thesis explains the type of queries that can be handled (subproblem P1), and the design and implementation of the network (subproblem P2). To measure its performance, multiple tests are created. These are described in this document, along with the results (subproblem P3).
1.4 Thesis outline

The global problem space is now clear, but before we dive into the rest of the thesis, we give a brief outline of what to expect.

Chapter 2 discusses all the concepts that one should be familiar with to understand this thesis. Reading and understanding this chapter should make the thesis self-contained. Chapter 3 walks through the most important design decisions that were made in this project, and discusses the arguments upon which the decisions were built. Chapter 4 describes the actual implementation of the design described in its previous chapter. Chapter 5 describes the setup of all the tests that were done to determine the system’s performance, and Chapter 6 discusses the results of those tests. The final part of the thesis, Chapter 7 discusses the conclusions, also by looking back at the problems that were formulated in this chapter.
Chapter 2

Preliminaries

This chapter elaborates on techniques and definitions used in Linked Data, that can be used as a foundation for this project. Furthermore, we give an overview of the state of related research that has been done in the field of distributed (Linked Data) query processing.

We start with a description of Linked Data in Section 2.1, the method to publish the data that is used in this project, which is fairly different from traditional approaches of publishing data on the Internet. Next, discussed is how this data can be queried, in Section 2.2. In the last part of this chapter, Section 2.3, we dive into the latest research that has been done in a more specific querying approach, peer-to-peer systems.

After reading this chapter one should be sufficiently familiarised with the subject to read the rest of this thesis.

2.1 Linked Data

In this section we introduce Linked Data and the techniques upon which Linked Data is built, and how the World Wide Web can be evolved into the Semantic Web, by using Linked Data.

2.1.1 Motivation

The World Wide Web as we know it in a traditional sense, contains documents stored on web servers. Those documents contain human readable data and links to other documents of this kind. Since this Web was built for human consumption, it is hard for a machine to understand the meaning of this data.

For example, a document about the movie ‘Lost in Translation’, could contain an anchor to the document about the actor ‘Bill Murray’. Automating the process of retrieving all the actors that played in this movie is hard because a machine does not understand the difference between the anchors that depict an actor relation and anchors that depict for example a director relation. An actor would be confused with a director (in this case ‘Sofia Coppola’), see Figure 2.1.

2.1.2 Linked Data

The problem with the traditional World Wide Web is that data is presented in various - machine-unreadable - content formats, which are accessed using a wide variety of web APIs. When consensus is reached in the content formats to use and the APIs that are accessible, new possibilities open up.

A solution that gained a lot of attention in research in recent years, is to describe the content on the web using structured (machine-readable) metadata, ‘data about data’. Just as a library catalog would describe what is in the library, the metadata used here describes resources on the

---

1Source: Internet Movie Database (http://www.imdb.com)
web. By using machine readable metadata to describe resources and links between these resources, an extension of the Web with a global data space of Linked Data is created, called the Semantic Web. With Linked Data the Semantic Web can be created, upon which lots of new types of applications can be built.

Using the example from the previous section, the connection between two resources, the movie ‘Lost in Translation’ and actor ‘Bill Murray’ is now described by a link (the movie:actor property) for which it is known that it depicts an actor relation (which is a resource of its own), as is visualised in Figure 2.2. Since we know how an actor relation is described, we could easily automate the process of finding all the actors playing in this movie.

A good overview of the history and current state of the art of Linked Data is provided in [12, 33].

2.1.3 Technologies used in Linked Data

The technologies on which Linked Data builds are the Unified Resource Identifier (URI) [42], the HyperText Transfer Protocol (HTTP) [24] and the Resource Description Framework (RDF) [38], which will each be discussed in this section.

The two technologies behind the traditional World Wide Web are URI and HTTP. URIs are more generic than the well-known URLs. While URLs are used as addresses for entities – for example on the Web – and therefore serve as both a locator and identifier, URIs identify entities but not necessarily locate them. URLs are thus a subset of URIs.

HTTP is a protocol built upon the Transport Communication Protocol (TCP) [17], and used to transfer resources that can be serialised as a stream of bytes. Retrieving such a stream of bytes is possible by dereferencing a URI that starts with http:// (which denotes the HTTP protocol).

The traditional World Wide Web uses URIs in combination with HTTP to retrieve published information such as documents (for example in HTML) without any restrictions on the structure of those documents. In Linked Data, resources are identified by a URI, and the metadata describing those resources adhere to a specific data model: RDF.

The idea behind RDF is that it is a data model for representing named properties and their

---

2 http://semanticweb.org/
3 Source: Internet Movie Database (http://www.imdb.com)
values. The model, as described in [38], consists of three object types: resources, properties and statements.

A resource is what is being described by RDF expressions, and is identified – as discussed – by a URI. A resource can be anything, accessible or not accessible via the Web. A property says something about a resource, for example a relation or an attribute, and a statement is what puts it all together, a specific resource with a named property and the value of this property. In an RDF statement – also called an RDF triple – these three parts are called respectively subject, predicate, object, as depicted in Figure 2.3. Using our running example we can now describe the relation between the movie ‘Lost in Translation’ and actor ‘Bill Murray’ as an RDF triple, visualised in Figure 2.4.

The predicate or property can be a resource of its own, identified by a URI and having its own properties. The object (the value of the property) can be a resource identified by a URI, a literal (i.e., a string or other primitive datatype) or a blank node. A blank node – also known as an anonymous resource – is a resource for which no URI or literal is given.

In the next section the informal description of Linked Data and RDF is formalised.

Formal definitions

In literature, the notions of an RDF triple and a Linked Data Resource are usually formalised as in [6, 29, 31]. In this section these formalisations are summarised, starting with the RDF triple and RDF graph:

**RDF Triple.** Given the set of URIs $U$ and the set of literals $L$, $(s, p, o) \in U \times U \times (U \cup L)$ is an RDF triple. Here, $s$ is the subject, $p$ the predicate, and $o$ the object of the triple.

**RDF Graph.** A finite set of RDF triples is called an RDF graph.

HTTP URIs are used as URIs in Linked Data (as will be briefly explained in the next section), which upon retrieval return a description of the resource identified by the URI, in the form of RDF triples. Each URI can therefore be seen as a Linked Data resource of RDF triples. The following definition formalises the Linked Data resource and graph:

**Linked Data Resource.** A Linked Data Resource $d = (T^d, URI^d)$, such that:

1. $T^d$ is an RDF graph; and,
2. $\forall t \in T^d : s(t) = URI^d$, where the function $s(t)$ returns the subject of the triple $t$; and,
3. $URI^d \in U$. 
Some definitions of a Linked Data Resource also include triples in the RDF graph which have the object equal to the URI of the resource. For simplicity, we restrict ourselves to the definition above, however.

**Collection of Linked Data Resources.** A collection of Linked Data resources is a collection of metadata – or catalogue data – held by a collection holder (as introduced in Chapter 1).

With the definitions in place, we can understand the exact meaning of Linked Data in the context of this project. Next will be described how to publish Linked Data.

### 2.1.4 Publishing Linked Data

In order to create a global data space, the so-called Web of Linked Data, some issues need to be addressed. These design issues are explained in this section, and can be seen as four steps one should take into account when publishing Linked Data on the Web. These steps are also known as the Linked Data principles, listed as follows:

1. Use URIs to identify the entities described by the published data. Use common vocabularies as much as possible (for instance FOAF, SIOC and Dublin Core) and only if they do not provide the required terminology, a publisher should create a new vocabulary (as specific as possible). More details on this are also available in .

2. URIs should be dereferencable via HTTP, so that people can lookup URIs. Also the URIs that identify terms in a vocabulary made by the publisher himself should be made dereferencable, so that the vocabulary explains itself.

3. By dereferencing a URI one will be provided with descriptive information about the URI, either by using its SPARQL endpoint – which will be discussed later – or by just retrieving serialised RDF triples directly.

4. Descriptions (such as retrieved RDF triples) of URIs should contain links to other resources using their URIs, this way data resources become connected and a global data space is created.

As mentioned in step 3, RDF triples need to be serialised in order to easily retrieve them. Different serialisation formats exist, of which a few are listed below.

- RDF/XML. An often used serialisation format where RDF is implemented in XML.
- Turtle. More suitable than XML in cases where human inspection is necessary, due to its readability.
- RDFa. Here RDF triples are embedded in HTML, so that the page’s original layout and styles are maintained. This way the document is both human- and machine-readable.

### 2.2 Querying Linked Data

Now that the foundations of Linked Data are described, it is time to discuss querying Linked Data. We summarise the research that has been done on distributed query processing in general, and of Linked Data in particular, to be able to make well-founded design decisions in Chapter 3.

---

4 [http://www.foaf-project.org/](http://www.foaf-project.org/)
5 [http://sioc-project.org/](http://sioc-project.org/)
6 [http://dublincore.org/](http://dublincore.org/)

---

Peer-to-peer Search over Linked Data in the Research and Education Space 7
2.2.1 Traditional approaches of querying distributed data

Before diving into querying distributed Linked Data resources, we briefly look at the general challenges of querying distributed data, since approaches on this (more general) domain could be relevant to the more specific field of Linked Data. As described in Section 2.1, in the Semantic Web, Linked Data resources are connected. Therefore, querying multiple (distributed) resources is required when querying Linked Data, which leads to new challenges.

Querying distributed data is a long studied subject in the literature. A great amount of literature is available, addressing the challenges for the general problem of querying distributed data and the associated pitfalls [11, 39]. An interesting theory about the trade-offs that need to be made in distributed systems is the CAP theorem, described in [14]. The CAP theorem states that a distributed system can only provide two of the following three properties: Consistency (every part of the system sees the same data at the same time), Availability (every request to the system gets returned with either a failure or a success response), and Partition tolerance (the system is able to process requests even when part of the system is unavailable). All of this literature originates from the research done in the context of distributed and federated query processing, but it applies to Linked Data as well.

When considering the Web of Data however, new challenges arise. Data is machine-readable, but not structured as we know data from relational databases. Query execution cannot be assumed deterministic anymore and universal cost models become outdated. These and more – often underestimated – challenges are pointed out in [59]. In [60], the ACE theorem – inspired by the CAP theorem – describes the relationship between Alignment (the query results are up-to-date), Coverage (all sources are considered for the query) and Efficiency (the query engine runs with bounded resources). The authors state that these properties are functions of each other, and therefore cannot be controlled all together.

2.2.2 Querying multiple Linked Data resources

The most basic form of search in Linked Data is to perform full-text search on literal object parts of triples, and boolean (or even conjunctive) search on predicates and non-literal objects. Although this type of search is not specifically designed for Linked Data and does not make use of its full potential, it is straightforward to implement.

The current, more sophisticated, standard in Linked Data query processing however is SPARQL, of which the syntax, semantics and complexity are described in [5]. The latest version of SPARQL is described in [6].

Based on the definition in Section 2.1.3, a resource is identified by a URI and thus, every URI in a domain is a separate resource. SPARQL is often used to query over all the URIs belonging to that domain, using an endpoint like for example in the Linked Movie Database. This means that in the entire Semantic Web, consisting of many collections, the SPARQL query engine retrieves results from just one collection of resources.

Querying multiple collections of Linked Data resources is a rather different topic. Even though the subject of querying multiple (distributed) Linked Data resources is a subset of querying distributed data in general (as discussed in the previous section), many different query approaches exist in the literature. Two good overview papers summarise and categorise the approaches to which the majority of research is conducted [30, 32]. The most important dimensions taken into consideration for examining the approaches are:

- **Data resource selection: index-based, live-exploration or hybrid**
  How are data resources selected during query execution, are they selected from an index, are they discovered ‘live’ during execution by dereferencing URIs or both? What are the consequences with respect to making use of the full potential of the Web of Linked Data?

- **Supporting data structures**

  [http://data.linkedmdb.org/sparql](http://data.linkedmdb.org/sparql)
What data structures support the query execution process? Examples are caches, statistics or indexes.

- **Data resource ranking**
  If data resource ranking is applied, resources are ranked before they are processed, for example based on relevance or access times. Ranking could also occur on individual triples from sources, instead of the source as a whole.

- **Integration of data retrieval and result construction**
  In the more traditional approach of query execution, data is first retrieved and the results are constructed afterwards. By integrating these two phases, results can be presented while data retrieval is still ongoing.

- **Response time**
  The time after which the first result becomes available.

- **Execution time**
  The time after which all results are available, and thus query execution is completed.

- **Freshness**
  How up-to-date are the query results?

The following sections will discuss several high-level query approaches, classify them using the above-mentioned dimensions, discuss some general advantages and disadvantages, and briefly discuss the research that has been done in each approach.

### 2.2.3 Data warehousing

In this approach, data resources are collected into a centralised data repository, called a data warehouse. The simplest approach of collecting data is loading RDF dumps into the centralised database, but crawling the data or accessing SPARQL endpoints is also possible.

With the data warehousing approach the best performance is achieved, since at query execution time, the data is already retrieved from the distributed sources. Indexes might be available, or statistics about the data, calculated in advance.

The downside of this approach is that initialising and maintaining the centralised repository is a challenge at the least. It is extremely hard (or even impossible) to maintain a centralised database that reflects the entire Web of Linked Data, let alone maintaining such a database. Therefore, not all resources will be collected, and the freshness of the collected data is not satisfactory.

### 2.2.4 Federated query processing

The federated query processing approach assumes a selected number of data collections that share the same type of query endpoints, for example SPARQL endpoints. A mediator, also called a query federator, splits the user query into subqueries that are executed on each endpoint. The results are transmitted over the network and combined into one result set. Extensive research has been conducted, dedicated to the federated query processing of SPARQL endpoints [48]. Detailed approaches are given in [47, 53, 54].

The advantage of this approach is that it retrieves results directly from the original up-to-date data, which preserves freshness. The downside however is that each selected data collection is forced into providing a SPARQL endpoint, and maintaining this endpoint, which comes with a significant amount of effort. The query execution is much slower than in the data warehouse approach since all data has to be transferred over the network. Most important of all is the fact that due to the above-mentioned disadvantages, not all data resources will be considered for a query, thus as with the data warehouse approach, the full potential of the Web of Linked Data is not used.
2.2.5 Active discovery based query federation

A solution to the problem of not using the full potential of the Web of Linked Data is in part provided by the active discovery based query federation approach. Here, in addition to standard federated query processing, unknown but potentially relevant resources are discovered during query execution. These discovered sources should all have SPARQL endpoints, which is the downside of this approach.

Although freshness is maintained and more Linked Data data resources are used, in this approach the full potential of the Web of Linked Data is not used either, since not all resources available can be expected to provide a SPARQL endpoint.

This approach is proposed in [32], but we are not aware of any systems implementing this idea.

2.2.6 Link traversal based query execution

Link Traversal Based Query Execution (LTBQE) is an approach in which the exploration of data resources by dereferencing URIs (‘link’), is intertwined with the construction of the result of the query. Since for each data resource no SPARQL endpoint is required, the query execution system relies solely on the Linked Data principles described in Section 2.1.4.

Since the Web of Linked Data is unbounded, it is unfeasible to explore all available data resources and create one huge index before generating results. Thus, a query will be executed on live data, therefore this approach works well with respect to freshness and using the full potential of the Web of Linked Data.

The problem here is a reasonable response and execution time. However, by using integration of data retrieval and result construction a result could be available as soon as one is found, because of the live-exploration approach it still has to be retrieved over the network. The execution time is an even greater challenge: since theoretically all sources will be considered for a query it is not that obvious when to stop exploring new sources.

Detailed approaches to address these challenges are given in [29, 31].

2.2.7 Peer-to-peer query execution

Peer-to-peer query execution is an approach not specifically designed for Linked Data, however this approach can be useful.

Data resources can be indexed using a data structure that is distributed over all the peers, or stored at each peer locally. To make sure response and execution times are reasonable, the network can be structured based on the semantics of the query, creating a semantic overlay network. This kind of structure will make the system more scalable than an unstructured network.

The downside of this approach is that the full potential of the Web of Linked Data is not used, because a copy of the original data has to be made as an index to make it available in the network of peers. The freshness therefore depends on how up-to-date the index is.

A survey of systems implementing this approach is given in [4, 25], and the next section discusses peer-to-peer querying in more detail.

2.3 Peer-to-peer querying

Apart from being applied to Linked Data, peer-to-peer systems are well-known and well-studied in research, and have had many applications in the past. Peer-to-peer systems can be unstructured or structured, either by using a distributed hash table (DHT) to make a structured overlay network, or the semantics of the query to create a semantic overlay network (of which a general study is done in [20]), or a combination of both.
2.3.1 Traditional peer-to-peer systems

Examples of unstructured systems are Gnutella [26] and Napster [43]. Since they have no structure, it is easy for peers to join and leave the network (the churn rate can be high). However, the downside of the network having no structure is that querying it for a specific piece of data results in flooding the network, sending the query to all nodes in the network because there is no knowledge of where the requested data is held. This makes the system incapable of scaling very well.

In peer-to-peer networks structured with a DHT, each peer gets assigned a piece of the hash ring. This creates a structured overlay network so that it is no longer necessary to query all peers for a piece of data. It handles the flooding problem of unstructured peer-to-peer networks, but the query functionality is still very basic. BitTorrent [19] is a popular example of this type of network. In research, Chord [57], P-Grid [1] and Pastry [51] are well-known examples.

2.3.2 Modern peer-to-peer networks

More recent research has been conducted on how to expand the query capabilities, while keeping the network structured and the performance reasonable. A distributed data management infrastructure can then be created, on top of a peer-to-peer network.

These networks use the semantics of the query. RDFPeers [16, 15] for example, uses a DHT to relocate the data and query semantics to process queries. It can be seen as a distributed RDF triple store, which distributes all the triples in a DHT based on Chord, and is because of that able to handle range and multi-attribute queries. MIDAS-RDF [58] and [50] use a similar approach. Also in this category is Atlas [36, 34, 35], which stores data on top of the Bamboo DHT [49], and even provides full SPARQL query language functionality.

Edutella [18, 44, 45] is a peer-to-peer query infrastructure that uses a common query model for all the peers. It expects peers to have a wrapper to translate the common model into their local query language, creating a semantic overlay network. Underneath this overlay, an overlay of super peers is used to enhance performance, which however introduces the single point of failure problem. Edutella is built upon JXTA [27], a framework for building peer-to-peer applications.

GridVine [21, 2] is similar to Edutella in the sense that it separates a physical and logical layer, the latter in which a semantic overlay network is created. Here it supports semantic interoperability between peers and inheritance and mapping of schemas. The physical layer is a structured overlay network provided by P-Grid.
Chapter 3

Design

Now that the problem statement is clear, and relevant directions of research have been discussed in the previous chapter, it is time to look at the design of the system, and therefore addressing subproblems \( P_1 \) and (partly) \( P_2 \).

We start by stating the requirements and assumptions, in Section 3.1. With those requirements in mind, we look whether the research and concepts described in Chapter 2 are applicable to the system that we are designing, and why. The global design decisions will be explained in Section 3.2 and more details of the design are discussed in Sections 3.3 and 3.4. The remainder of this chapter, Section 3.5, focuses on the expected time-complexity and potential bottlenecks that come with the presented design.

3.1 Requirements and assumptions

In consultation with the BBC, the following requirements and assumptions have been set. For the system to be successful, these must be met. The requirements are listed below, split up into ones that are absolutely required (must-haves) and ones that are preferred but not an absolute necessity (nice-to-haves):

**Must-have:**

R1 Queries support full-text search in the literal object of an RDF triple

R2 Queries support conjunctive search in the predicate/object pair of an RDF triple, where the object is a non-literal

R3 Queries that are formed of a combination of queries as described in R1 and R2 will be executed as conjunctive queries

R4 Query results should be subject URIs that match the query

R5 A collection holder should not have to modify their data specifically for using our system

R6 The system should be fault-tolerant in the sense that the unavailability of one collection should not lead to the unavailability of another collection

R7 The system should be scalable, meaning a network or cluster doubled in size, should show a sub-linear increase in query execution time

**Nice-to-have:**

R8 A collection can be easily added or removed from the system
CHAPTER 3. DESIGN

R9 Usage of existing (open source) solutions to the extent possible is preferred
R10 Query execution time preferably does not exceed the order of seconds
R11 The system should be able to handle a throughput of at least 100 queries per second, running on one standard 2015 non-SSD dual-core machine per collection on average

The assumptions are listed below:

A1 The collection and therefore its data resources will adhere to the Linked Data principles as discussed in Section 2.1.4
A2 A single collection of data resources can contain up to 25 million RDF triples
A3 A collection holder is able to run any Linux-based provided application, either within existing hosting facilities or by procuring new facilities, such as dedicated servers or virtual machines

3.2 High-level design

The idea behind RES is that an API abstracts collections of metadata from multiple collection holders. Thus, at the highest abstraction level, there are two approaches we could follow to design such an API: a centralised and a decentralised, distributed approach.

3.2.1 Centralised

Although the centralised approach is very straightforward – many off-the-shelf products can provide a solution – it has a number of drawbacks. First, all data needs to be ingested by one central data warehouse, as described in Section 2.2.3, this is a very time- and resource-consuming task. Then, if in the original data (hosted in the collection holders’ data centre) a change is made to one of the data resources, this ingestion process needs to be redone, at least partly. The time between the original data change and the update of the data warehouse results in a lower data-freshness of the warehouse.

Besides the drawbacks of getting the system up and running, and keeping it up-to-date, there are other even more important drawbacks. When the number of collections that are ingested grows, the data warehouse can potentially become huge. This will likely impact the query execution time negatively, preventing the system from scaling, and will require a great amount of resources in one single place, resulting in a single point of failure. It is clear that maintaining this data warehouse is a complex task, both technically (to fulfil for example requirements R6 and R7) and non-technically. This means that one central place should be maintained by multiple collection holders, each of them providing part of the resources, clearly not ideal.

3.2.2 Decentralised

Possibly the first step in turning any centralised data warehouse into a more scalable system is dividing it into smaller pieces, making it distributed, but within the same data centre. Although this is a common solution, in our use case it still does not perfectly solve the problem. Data freshness remains questionable and maintaining the system technically is more feasible, it would scale better. The problem is still requirements R6 and R8, and managing the non-technical part as described above remains far from ideal. This is why for example Solr [55], SIREn [23] and Sphinx [56] are not appropriate solutions, the data management of multiple collection holders is complex.

In a truly decentralised, distributed approach, where we make use of assumption A3, each collection holder could provide the resources for a part of the network, and would be responsible for his own data in the system. The more collection holders and data collections there are, the more resources will be available. Keeping reasonable query times can be a challenge, but it solves both
A decentralised and distributed approach is clearly in favour, but there are still a lot of choices to be made. In the field of distributed query processing, one approach is federated search (see Section 2.2.4), but due to the (potential) scale of the system, multiple layers of federators might eventually be needed, which could slow down the system. Besides, the concept of a federator in federated search is a single point of failure: it routes the queries from one federator, perhaps via intermediate layers of federators, to the relevant nodes. Even when replication is used to solve that problem, who will be responsible for the federators?

3.2.3 Peer-to-peer

Another popular approach, discussed in Section 2.3, is peer-to-peer search. Each peer is equal in terms of functionality and resources it provides, and responsibilities it has in the network. In our use case, every peer hosts its own data, which makes it easy for peers to join and leave the network (in accordance with requirement R8). Each peer, or node, has the functionality of processing a query, retrieving results from its own data or routing the query to another node that might have results. Thus, there is no single point of failure (meeting requirement R6), which makes peer-to-peer search the preferred choice for our system.

A very brief overview of the approaches that we considered is visualised in Figure 3.1. With the chosen peer-to-peer network, we now have a system that is run on multiple peers, that is fault-tolerant (one peer being unavailable does not bring down the entire system), and that has the potential to scale.

With this high-level design in place, it is time to look at some details, to make sure that the design as a whole has the potential of meeting all the requirements. In Section 3.3 all the details of designing the network are discussed and Section 3.4 describes the design of a single node in the network.
3.3 Design of the network

Before we design the network in more detail, it is important to know what a query looks like and what it means to query. This is described in the next section.

3.3.1 A query

A query sent to the network, is – by requirements \( R_1 \) and \( R_2 \) – defined as follows:

\[ \text{keyword}, (\text{predicate}, \text{object}), \ldots \]

A query can contain either a keyword, a predicate/object pair, or both, called query parts. A query containing multiple predicate/object pairs is also possible. The query parts form – by requirement \( R_3 \) – a conjunctive query. Examples of queries (one on each line) are:

- "King of Pop", (\( \text{rdf:type}, \text{foaf:Person} \))
- (\( \text{rdf:type}, \text{bibo:Book} \)), (\( \text{dcterms:language}, \text{http://lexvo.org/id/iso639-3/eng} \))
- "Shakespeare"
- (\( \text{event:place}, \text{http://bnb.data.bl.uk/id/place/Eindhoven} \))

A query can be retrieved by a node in the network, which contains a collection of data resources. In accordance with requirement \( R_4 \), a URI of a data resource from that collection is returned as a result if for each predicate/object pair in the query a triple exists in this resource in which the predicate and object match. Furthermore, if a keyword is present in the query, there should exist a triple in this data resource with a literal object that ‘contains’ the keyword. Common information retrieval techniques can be applied (like tokenisation) to determine whether a keyword is contained in a string of text (a literal object in our case).

We assume that a query has at most one keyword query part. Because it is a full-text search term, more than one actual word can always be used as a search term.

In the absence of information on where to route a query, we don’t know if the node that received the query, or any other in the network, has data that is relevant to that query. The network is unstructured, so the only option is to send the query to all the nodes in the network, which results in flooding. Eventually the network will get flooded by queries sent to nodes, most of which don’t even have results for the queries sent to them. Ideally we want to structure the network so we know upon arrival of a query, which nodes to send it to.

3.3.2 A structured network

To make the network structured we could use a semantic overlay network as discussed in Section 2.3. Based on the semantics of the query, we could send it to a specific node. However, the fact that the semantics of the queries are just a keyword and a predicate/object pair does not make structuring the network straightforward. Furthermore, the data collections must then be shipped between nodes to structure the network, this would make it hard to meet requirements \( R_6 \) and \( R_8 \).

Another possibility is to store all keyword and predicate/object pairs as keys in a structured overlay network, using a DHT which is distributed over all nodes in the network, and nicely splitting the structured overlay from the actual data and thus fulfilling requirements \( R_6 \) and \( R_8 \). The value of those keys is a list of nodes that have results for that keyword or predicate/object pair. Indexing this information in advance though is resource intensive, and the index might become too big. Furthermore, probably the majority of the index will never be used, since lots of keywords are not used in queries. The advantage on the other hand is that the index will be accurate.
Our approach

Our design uses a combination of structured and unstructured approaches. Starting with an unstructured network, the system structures it when queries are executed, by indexing for that query the nodes that returned results. This will be slower for new queries, but no resources are wasted on indexing keywords that are never used. The trade-off is that if a query is executed that does not hit the index, it will take a bit longer and will waste some resources, but that will only happen once. Next the details of this process are described.

3.3.3 Storing routing information

When a query enters the network, it is split into query parts, the keyword and predicate/object pairs that make up the query (as discussed in Section 3.3.1). For instance, the query "King of Pop", (\texttt{rdf:type, foaf:Person}) is split into "King of Pop", the keyword, and (\texttt{rdf:type, foaf:Person}), the predicate/object pair.

When a node routes a query to other nodes, it waits for each node to return a set of results, before storing routing information into the DHT. This is the information that links a query part to a particular node, suggesting that node has results for that query part.

There are a few possibilities when storing the routing information in the DHT. If the result set returned by a node is not empty, we can be sure that each query part exists in that node’s data (queries are conjunctive). Thus, the node can be added to each query part key in the DHT for this query. If the result set is empty, the situation is a little more complicated. Although we can be sure that at least one of the query parts do not exist in the node, we are not sure about the others. This is why the node should give an extra signal on which query part did return results, if any. That way we can still update the DHT and profit from the stored information later.

For each node to be aware of all the other nodes in the network, each node registers himself in the network by placing his identifier, address and timestamp of when it was last updated, into the DHT. This only happens once, and as soon as the node changes its data the timestamp is updated, so that node timestamps can be compared to the routing information. This design supports nodes entering the network, their timestamp will be newer than any of the routing information stored in the DHT at that point. The design can easily be extended to let nodes leave the network, or be (temporarily) offline.

3.3.4 Using routing information

Using the routing information is done according to the pseudocode listed in Algorithm 1. The algorithm checks for each query part if it exists in the DHT (line 6) and if not, that query part has apparently never been queried before. We then do not know which nodes contain relevant results, and therefore query them all (line 15). If the query part does exist, we retrieve the set of nodes that are stored in the DHT (line 7) and next check for each node if it is newer than the current query part that is being checked (line 10). If so, this node should be added to the list, because it could contain new results, or not have results anymore.

After obtaining a set of nodes from the first query part, we can directly assign it as the definitive set of relevant nodes (line 19), with the sets of nodes from the subsequent query parts, we perform an intersection (line 21). By doing so, part of the query processing is done at this stage. For instance, given the above mentioned query, if node $x$ contains (\texttt{rdf:type, foaf:Person}), but does not contain "King of Pop", we can already be sure that this node will never return a result for this query, due to the conjunctive nature of queries. We skip node $x$ and save resources.

Either after the definitive set of nodes turns out to be empty (line 24), or we checked all the query parts, the algorithm is finished.

3.3.5 Distributed hash table settings

Storing and using routing information requires a DHT. DHTs come in all shapes and sizes, but it usually boils down to how it uses the CAP theorem, described in Section 2.2.1.
1: nodes ← ∅
2: 
3: for all query_part in query do
4:    partial_nodes ← ∅
5:    
6:    if DHT_KEY_EXISTS(query_part) then
7:       partial_nodes ← DHT_GET_NODES(query_part)
8:    
9:       for all node in (all nodes in network) do
10:          if DHT_TIMESTAMP(node) > DHT_TIMESTAMP(query_part) then
11:             partial_nodes ← partial_nodes ∪ node
12:       end if
13:    end for
14:    else
15:       partial_nodes ← (all nodes in network)
16:    end if
17: 
18:    if nodes = ∅ then
19:       nodes ← partial_nodes
20:    else
21:       nodes ← INTERSECT(nodes, partial_nodes)
22:    end if
23: 
24:    if nodes = ∅ then
25:       break
26:    end if
27: end for

Algorithm 1: Finding relevant nodes
In our design a DHT that focuses on availability and partition tolerance is preferred. The table not being consistent at all times is not a problem, the worst-case situation is that routing information is present in the DHT but due to its inconsistency, a node is unable to find the information. Therefore, the node will send the query to all the nodes in the network, but will still get the correct results in the end.

Next, the node will try to update the information in the DHT, which will be the same as the information already present. Although some resources will be wasted when this happens, the benefit is that the consistency property of the DHT can be relaxed (classifying the DHT as ‘eventually consistent’).

All the aspects that are discussed in this section can be implemented in two applications. First, a server application that runs on every node. We will call it the routing application from now on, and its responsibilities are receiving queries, retrieving routing information, routing queries to the right nodes, receiving and combining the results, and updating the routing information. The second application is a DHT that provides the storage for the routing information. In Section 4.2 of the next chapter, the implementation details of these two applications are discussed.

Figure 3.2 summarises the design discussed so far. When a query is sent to the network (1), first the routing information is retrieved from the DHT to find relevant nodes (2), then queries are routed to and executed on the relevant nodes (3), after which results are sent back to the node that received the query initially (4). If new routing information is available, it is stored in the DHT (5), and finally the combined results are sent back to the client library (6).
3.4 Design of a single node

We will now look at the design of a node. This is where a query ends up after being routed by the routing application, and the actual search is being performed.

3.4.1 The database

According to assumption A1, a collection holder provides his data in Linked Data format, published online or in a data dump. Searching in data resources that are published online (for example with LTBQE, as described in Section 2.2.6) is a bad choice in terms of query execution time. We can do better because – as opposed to LTBQE – there is no ‘zero-knowledge’ constraint, the data we want to search is known in advance.

Using a SPARQL endpoint is neither an option, as we cannot be sure that a collection holder provides it. Even if a collection holder does, the SPARQL query language is not capable of performing full-text search operations, it only supports basic string functions [28]. Hence, the preferred option – and meeting requirement R5 – is to gather the data of a collection holder in one place by crawling or using a provided data dump. Searching through the raw gathered information is still time consuming, so we need to speed this process up by preparing the data for searching.

Many off-the-shelf products exist that can help with this process, which also provide full-text search functionality. Our requirements however, with full-text search on one part of the data (the literal objects of triples) and searching for exact matches on predicate/object pairs on the other part, are very specific. We did some initial tests using a single off-the-shelf product to meet our search requirements, but found that it was rather hard to configure such a product to exactly meet those requirements. We therefore decided to go for a more low-level design, in which the database splits the data into two parts, and uses two off-the-shelf indexes to search the data: one that is great at full-text search, another that is great at simple lookups. For the latter one, a key-value store can be used, where the key will be the predicate/object pair of a triple and the corresponding value the subject of that triple. Instead of sticking to a single, high-level off-the-shelf product for all the search functionalities, we now have a more flexible design.

When the database receives a query, each of its query parts are queried in their respective index. Next the actual result retrieval starts, which follows the well-known sort-merge-join algorithm [63]. We assume the indexes to return sorted results, so that no separate – and costly – sort step has to be included before joining the results. We therefore only have to merge-join the results, according to the pseudocode listed in Algorithm 2.

As soon as one of the indexes runs out of results, we can stop (lines 10 and 18). If we find a match, we add it to the result list (line 23), and otherwise we compare the results and – based on their ordering – we fetch a new result to compare from either one of the indexes (line 24).

As a summary of the above, Figure 3.3 illustrates the query execution process. The database described in this section can be easily implemented in one server application, that runs on a node alongside the routing application and the DHT. The implementation is discussed in Section 4.1 of the next chapter.
1: \(result \leftarrow \emptyset\)
2: \(current_a \leftarrow \text{null}\)
3: \(current_b \leftarrow \text{null}\)
4:
5: \textbf{while True do}
6: \quad \textbf{if} \(current_a = \text{null}\) \textbf{then}
7: \quad \quad \textbf{if} \ \text{INDEXA\_HASNEXT}( ) \textbf{then}
8: \quad \quad \quad \text{current}_a = \text{INDEXA\_NEXT}( )
9: \quad \quad \textbf{else}
10: \quad \quad \quad \textbf{break}
11: \quad \quad \textbf{end if}
12: \quad \textbf{end if}
13:
14: \quad \textbf{if} \(current_b = \text{null}\) \textbf{then}
15: \quad \quad \textbf{if} \ \text{INDEXB\_HASNEXT}( ) \textbf{then}
16: \quad \quad \quad \text{current}_b = \text{INDEXB\_NEXT}( )
17: \quad \quad \textbf{else}
18: \quad \quad \quad \textbf{break}
19: \quad \quad \textbf{end if}
20: \quad \textbf{end if}
21:
22: \quad \textbf{if} \(current_a = current_b\) \textbf{then}
23: \quad \quad result = result \cup current_a
24: \quad \textbf{else if} \(current_a < current_b\) \textbf{then}
25: \quad \quad current_a \leftarrow \text{null}
26: \quad \textbf{else}
27: \quad \quad current_b \leftarrow \text{null}
28: \quad \textbf{end if}
29: \textbf{end while}

Algorithm 2: Performing a merge-join of two indexes
3.4.2 Ingesting the data resources

One important thing that still needs to be done before the system design is complete, is making sure the indexes get filled with data resources. In terms of design, this will not really be a challenge, the process is independent the system’s normal operation (processing queries), so the time it takes to index data is not high priority, as long as it is done correctly.

One simple application will basically be responsible for parsing RDF triples, preferably supporting multiple serialisation formats (as in Section 2.1.4), and separating the parsed triples into ones that have literals as objects, and ones that don’t. As already discussed, the former will be indexed in a full-text index, the latter in a key-value store. The implementation details are described in Section 4.4.

The remainder of this chapter describes the expected performance, in terms of time complexity and expected bottlenecks.

3.5 Expected performance and bottlenecks

To estimate the expected performance of the database, we look at the most costly operations that must be performed: fetching results from the index and joining those results.

Indexes that qualify for this design usually have a B-tree or hash table as implemented data structure, for which the time-complexity of lookup operations is $O(\log n)$ and $O(1)$ respectively on average (with $n$ the number of records in the index). Other operations like insert or delete don’t occur during normal query execution, so can be omitted here. The hash table seems a lot faster, but performance can be much less predictable because it depends on the data and the implemented hash function. Worst-case performance could be as bad as $O(n)$, if all records are stored in the same bucket (which, though, is highly unlikely). The worst-case performance of a B-tree on the other hand, is still $O(\log n)$. A B-tree also returns results in sorted order, which is convenient for merge-joining the results. The preference is therefore to use a B-tree, and go for stability and predictable performance.

If results are available for the query, joining the results has time-complexity of $O(n)$, where $n$ is the smallest number of results that the indexes returned. Theoretically, it depends on the size of the indexes and the number of results which operation, the fetching or the joining, will be the bottleneck of the database, but assuming a worst-case scenario, it would be safe to say the join operation will take most time.

The most important operations that are performed in the network are: fetching routing information from the DHT, receiving results from the nodes, merging those results, and storing routing information in the DHT, if needed.

In a DHT operation it takes a certain amount of ‘hops’ to find the node which has the piece of the hash ring that we need for the operation. This process usually takes $O(\log(N))$ hops, where $N$ is the number of nodes in the network.

Receiving results from the relevant nodes can be done in parallel, so the time it will take depends on the node that takes the most time to execute a query, which in turn – as we just discussed – depends on the number of results to be retrieved from the indexes. Merging the results from multiple nodes into one result set is an operation that needs to run through all the results ($O(n)$, with $n$ the number of results for a node), but since data is already received this operation will be very fast, the merging can even be done when results are still being received from another node.

Depending on the number of results for each node, and thus how long it will take to receive the results from all nodes, the process of receiving results from nodes is probably the bottleneck. This operation will be fast if no results are available, especially when this is determined from the routing information (and no node actually needs to be queried), in that case the using the DHT will be the bottleneck.
Chapter 4

Implementation

This chapter describes the implementation details of all the applications that were designed in the previous chapter, thus addressing subproblem $P_2$. We will motivate and discuss the libraries and off-the-shelf applications used to implement the system, and which measurements are taken to make sure the applications will have the required performance. In our description, we give a high-level overview of the internals of each application and the interaction between them. Details about specific software version can be found in Appendix A.

The database application and indexer are described in Section 4.1, and the DHT and routing application in Section 4.2. The last part of the chapter looks at the implementation of the protocol that is used for communication between the database and routing applications.

4.1 The database application

The core of the database application are the indexes in which the triples are stored, one for full-text search and one for searching predicate/object pairs, as discussed in Section 3.4.1. The preference was to use open source software to the extent possible (requirement R9), so a lot of those were looked at closely, before deciding to use one in the implementation.

4.1.1 Full-text search with LuceneClient

For the full-text index, we considered Lucene [41], Solr [55] (as non-distributed database), Sphinx [56], and Xapian [61]. All provide full-text search and could do the job. Solr is built on top of Lucene, so uses its functionality plus a lot more. Our use case however is rather simple so we need the least amount of functionality to do the job, thereby minimising overhead as much as possible. As for Sphinx and Xapian, they seem to provide similar functionality as Lucene, but we decided to go with Lucene due to its popularity. It is implemented in Java and provides a well-documented and simple to use API.

To search in a Lucene index, the LuceneClient class is implemented. The init() function initiates the static part of this class, creating a IndexSearcher object from the Lucene library, which is shared between all LuceneClient objects. When a query is sent to a LuceneClient object (using the query() method), it is parsed and then analysed with Lucene’s StandardAnalyzer (further discussed in Section 4.4.3). The parsed query, conceived as a Query object, is submitted into the – thread-safe – IndexSearcher object’s search() method, along with a maximum number of results, and a SortField object on which to sort the results (in our case, the subject URI). The search() method returns the most relevant results for the parsed query, in sorted order.

Fetching the actual results is done in the hasNext() method of LuceneClient object. Here the IndexSearcher is used to obtain a Document object with a stored subject URI (this object is more thoroughly discussed in Section 4.4.3). The next() method of the client actually returns the URI, the hasNext() method only checks if there is a next result available.
4.1.2 Predicate/object search with BerkeleyDBClient

As designed in Section 3.4.1, the predicate/object index should be a key-value store, because we don’t need any complex query capabilities, just scalable basic lookup and insert operations are sufficient. We looked at Kyoto Cabinet [40], Redis [52], and Berkeley DB [46]. Kyoto Cabinet does not seem to be maintained anymore, which makes it a bad choice, and while Redis is still in development, it has more capabilities than we need. We chose to go with Berkeley DB, it is known for its good performance and scalability, and has a Java API, which is convenient now that we decided to go with Lucene for the full-text index.

The BerkeleyDBClient class is implemented to provide search functionality in the Berkeley DB index. The static init() function opens the index, by creating an EnvironmentConfig object that allows error handling customisation and cache options, and a DatabaseConfig object is created, to define underlying data structure for this index. Berkeley DB supports both a B-tree and a hash table, we preferred predictability and decided to use a B-tree (see Section 3.5 for more details).

In the BerkeleyDBClient object, the static and thread-safe Database object is used to open a Cursor object (by calling the query() method). To put the cursor in the right position, the getSearchKey() method is called. The BerkeleyDBClient object’s hasNext() method will check if new results are available, by checking a MultipleDataEntry object for a new DatabaseEntry objects, which hold the data belonging to the search key. If the MultipleDataEntry object is exhausted, we try to fill it up first, before checking for new DatabaseEntry objects again. Like in LuceneClient, next() converts the current DatabaseEntry into an actual result.

The client classes have a different implementation, but a few implementation details are the same. These are implemented in an abstract class DBClientBase, which both index clients extend. This class provides functionality such as keeping track of number of hits for a query (numOfResults()), the number of results that have been returned (getCursorPos()), the last executed query (getQuery()), and whether there are results available (hasResults()). If we ever want to add a new index, all that needs to be done is extend the DBClientBase class with a new child class.

4.1.3 Query result construction with Intersect and DBInterface

Next, we need to combine the results that the index client classes generate, into one result set, using a sort-merge-join algorithm. This is what the Intersect class does, although the sort part of the algorithm can be left out, as both clients return results in sorted order.

Intersect can provide a few similar methods as DBClientBase does, namely hasNext() and next(). The implementation is different: hasNext() performs the join, and if that is successful, next() returns the result. To perform the join, Intersect needs two objects it can get results from, but not necessarily implementations of the DBClientBase class, it can also be another Intersect object, as long as the same interface is provided.

Like in the client classes, the process of retrieving results is split up in hasNext() and next(), because a Java function can only return results of one type, it is ‘strongly typed’. If we ask for the next result (in our case of type String) and it is not available, we don’t want to return some special String value, but rather the value false of type boolean. The most elegant way to do that is to split the implementation up into two functions.

The actual implementation of the merge-join in hasNext() follows Algorithm 2. Note that this function requires the result types to implement the Comparable interface, so that it provides a compareTo() method which can be used to compare the results. The Java built-in String type implements this interface by default.

As just mentioned, both Intersect and DBClientBase have to provide the same interface. We enforce this by creating an interface called DBInterface, which both classes implement. Now, a very basic query execution plan can be created: a tree consisting of Intersect objects as nodes, and actual indexes – DBClientBase objects – as leaves. This makes it very easy to extend the query semantics by creating a new class that implements DBInterface, for example Union, which
could implement the or operator as opposed to the and operator that Intersect implements. Just as with DBClientBase, implementation details are hidden behind interfaces, code is made modular and therefore easy to adapt or extend.

All classes described so far are visualised in Figure 4.1 along with their relations.

4.1.4 Query processing with DBClient

The DBClient class puts all the previous parts together, by organising it in its main function: query(). It first parses the query, serialised as json and passed as a parameter, into query parts, using Gson, a json library for Java. Next, it instantiates a list of LuceneClient and BerkeleyDBClient objects, choosing the right object based on whether the query part is a keyword or a predicate/object pair. With instantiating those objects, their query() method will be directly called, passing the query part as a parameter. Note that the first time the clients are instantiated (when the first query arrives) their init() function will run. This can take a significant amount of time, but will happen only once.

To set up the query execution tree, the static function get() from the Intersect class is called, which takes the DBClientBase typed list as a parameter, and returns an object that implements DBInterface, either an implemented DBClientBase object (if there is just one query part), or an Intersect object if there are multiple. get() is recursive, so that the length of the list does not matter.

At this point, it is irrelevant how exactly the tree is constructed, and of which objects it consists. Since it implements DBInterface, we can use the hasNext() method in a while loop as the condition and call the next() method each loop iteration to obtain the result. The tree takes care of the rest. Each result is put onto the ByteArrayOutputStream object, which was passed along with the query in the query() method of DBClient.

Now there are only two things left to do. First the cursors that are open in LuceneClient and BerkeleyDBClient objects need to be closed. Second, we need to know which of those objects returned results, because the routing application needs that information to update the DHT, as discussed in Section 3.3.3. Both tasks are done by running through the DBClientBase typed list we constructed earlier. DBClientBase provides a hasResults() function which solves the latter task, and the former is done by the close() method in LuceneClient and BerkeleyDBClient, which DBClientBase forces them to implement (it is abstract in DBClientBase).
In a LuceneClient and BerkeleyDBClient object, query() is called to open a cursor in the index and to set it in the right position, based on the query part. It basically does all the work before results can be retrieved. We tried to optimise DBClient and the way it calls those clients, by using threading. Opening cursors could be done in parallel, and results could be prepared before hasNext() is called. We found, however, that neither of these optimisations showed a significant performance gain and that furthermore, processing times became unpredictable. We decided therefore to run everything in DBClient consecutively.

The final piece of the database application, and therefore the final part of this section, is the actual server that listens to incoming connections and the handler that processes them as they come in.

4.1.5 Accepting connections with Server

The initialisation of the Server class creates a ServerSocketChannel object and binds a port to it. It also uses the Java built-in Executors factory to create an ExecutorService. The listen() method then enters a loop in which it listens for incoming TCP connections, and as soon as it accepts one, submits it – wrapped in a RequestHandler object – to the ExecutorService.

The ExecutorService is an interface with many implementations, each having different options so that the server’s behaviour can be configured extensively. It is for instance possible to create only a fixed number of threads, and put all the remaining tasks (accepted connections in our case) on a queue. The queue can be bounded as well, and tasks can have a maximum queue time after which they are discarded. These settings can thus put a limit on the amount of work a server will do, keeping it from overloading.

The implementation of ExecutorService that is used in Server is the CachedThreadPool. It keeps threads alive after usage (for 60 seconds to be precise) so that they can be reused, which is faster than creating new threads. Furthermore, when no connections are accepted, ultimately no threads are kept alive and resources are saved. The risk is that with the current settings an unlimited amount of threads can be created – and set to work immediately. The routing application is therefore responsible for not sending too many requests to the database application.

4.1.6 Handling routed requests with RequestHandler

Upon an incoming connection, a RequestHandler object is run in a separate thread. It uses the Network class as a wrapper for TCP communication. The RequestHandler class can process three commands: QUERY, RESULT, and PING (details on how these commands are created are discussed in Section 4.3).

The PING command is just to check if the server is still alive, and immediately sends back a response. When the QUERY command is received, along with the query itself, a new DBClient object is created, and its query() method is called with the query and a new ByteArrayOutputStream object as parameters.

After the query processing is done as described above, RequestHandler waits for the RESULT command. Although this is the only command possible now, it is implemented this way so RequestHandler can be easily extended with more commands later. After checking which indexes returned results, it sends a return signal along with the contents of the ByteArrayOutputStream object. Note that the results are sent all in one go. The return signal is one of following: NO_RESULT, LUCENE_RESULT, BERKELEYDB_RESULT or BOTH_RESULT. Note that when a query consists of two predicate/object pairs and one of them has results, the current signals don’t give enough information to tell which one of the two had results. For the current version of the system this will suffice.

The implementation of the database application is complete, the most important classes and their relation are illustrated in Figure 4.2.
4.2 The DHT and routing application

The routing application that we look into in this section, needs a DHT. Many ready to use DHTs are available, so instead of building our own, we use an existing one.

4.2.1 Riak as the DHT

For the DHT we considered Chord [57], Pastry [51], Tapestry [62], and Riak [37] (based on Dynamo [22]). We decided to go with Riak, because although it is less known, it is easy to set up, well documented and maintained, and has clients in many programming languages.

Riak is built to be ‘eventually consistent’ (see Section 3.3.5). The most important settings it provides us with are the storage backend, the ring size, and $R$, $W$, and $n_{val}$. The storage backend can be either set to memory or disk. The ring size is the number of partitions into which the table (the hash ring) is split. This value depends on the network size and will therefore be discussed in Chapter 5. $R$ is set to the number of nodes it needs to read a value from before the read can be considered successful. $W$ is set to the number of nodes to write a value to for a write to be successful, and $n_{val}$ is the number of replications that is created for each value.

In our setup, we use memory as storage backend and set $R$, $W$, and $n_{val}$ to 1, because the speed of read and write operations of the table is far more important than its consistency. To illustrate this, say for instance that it replicates each value three times, and two of those values need to be read or written for a read or write operation to be successful. Chances are that the overhead of using the DHT will then be larger than the performance gain we get from querying a few nodes instead of all the nodes in the network. This is of course a complicated trade-off which involves many factors and settings, especially since Riak can be extensively configured. We tried to use the most basic setup that will give us a decent performing DHT in terms of read and write operations.
4.2.2 Accepting client connections with ThreadedTCPServer

The routing application is implemented in Python, mainly to save development time. Python does a lot of heavy-weight lifting while maintaining a reasonable performance. A Python client is available for Riak, and we can build our own client to the database application.

The routing application is in essence a server that handles incoming requests in separate threads, analogous to the database application. This is the API to ‘the outside world’ – an outside client application in some programming language – which is run on each node. The server is implemented as a ThreadedTCPServer, which extends ThreadingMixIn and TCPServer, available in Python’s SocketServer library (note that multiple inheritance is possible in Python). The TCPServer class uses – as the name suggests – TCP, and is therefore capable of providing continuous streams of data between the server and its clients. The ThreadingMixIn class will make sure that when a connection is accepted, it is handled in a separate thread, therefore providing asynchronous behaviour.

When the ThreadedTCPServer is instantiated a RiakClient object is created, which wraps some custom methods around the Python client for Riak. If it is the first time this server runs, it updates its status via this object. More on the RiakClient class will be discussed in Section 4.2.5.

Upon an incoming connection, the request is handled by a RequestHandler object. This is where the main part of the routing application is implemented.

4.2.3 Handling client requests with RequestHandler

The RequestHandler class is capable of processing the same commands as in the database application (PING, QUERY, and RESULT), and also uses a Network class to communicate via TCP. In the current implementation the commands are not received in a loop, so after a PING or RESULT command the connection is automatically closed. It is straightforward however, to build a loop around this process and for instance use one connection for multiple QUERY commands.

When the QUERY command is received, the query is parsed first. Instead of using the rather resource-expensive json library in Python, we use a simple regular expression to obtain the list of query parts. With this list as a parameter, we call the get_db.clients() method of the RiakClient object, and find the relevant nodes to query (for details, see Section 4.2.5).

Next, each node is queried in a separate thread, using a DBClientConnection object: our client to the database application. It provides the connect(), query(), result(), and close() interface similar to those that most database clients provide. The DBClientConnection class also uses the Network class for communication, calling its connect() and close() methods to initialise a connection. Note that just as with the database application, all the results are retrieved in one go. This is done to simplify the result retrieving process. The results that are received from each node, are put in a result set held in the RequestHandler object, and is therefore shared by all the threads it created. After that the routing information is processed, which is discussed more elaborately in the next section. To manage concurrency correctly, each thread needs to acquire a lock before updating the main result set and updating the routing information. This is done using the acquire() method of the Lock object in Python’s threading library. The release() method makes sure the lock is free to use by other threads after this thread has finished.

4.2.4 Updating Riak

Processing the routing information follows the design from Section 3.3.3. If a node returned results, the process is trivial: all query parts in the query are updated to have results for this node. If a node returned no results, we use the signals and have four options.

If the node sent the signal NO_RESULT, then we know that none of the query parts returned results in this node and thus each query part in the query can be updated to have no results in this node. If the signal is LUCENE_RESULT, all query parts that are predicate/object pairs are updated to have no results in this node. Only if there is one query part that is a keyword, we can update that query part to have results in this node. If there are more, we don’t know which was the one
that returned results, so nothing is updated. If the signal is \textsc{berkeleydb}\textunderscore result, it is the exact opposite, all query parts that are keywords are updated to have no results and the query part that is the predicate/object type is updated to have results, but only if it is just one of that type. If the signal is \textsc{both}\textunderscore result, query parts are only updated if there is exactly one query part for each type.

To prevent all the threads from updating Riak separately, the information is gathered in a shared list, which holds for each query part the identifier of the nodes that did and did not return results. Concurrent writes to this list are managed as described in the previous section.

Next, we wait for each thread’s \texttt{join()} method to return, which makes sure all results from node that was queried are received and processed. Consecutively, the \texttt{update\textunderscore routing()} in the Riak client does the actual update and makes sure it does not write data back to Riak if nothing changed.

4.2.5 Retrieving and storing routing information with \texttt{RiakClient}

\texttt{RiakClient} is simply a class to have all Riak related functions in one place, which improves maintainability. Using another DHT instead of Riak is made easier too this way. Its initialisation starts with using the official Riak client for Python to make a connection to Riak. It offers ‘http’ and ‘protocol buffers’ as connection protocol. We use ‘protocol buffers’ since it is the fastest. Next we refresh the node list, so that we have a copy of all the nodes in the network in our application. The remainder of this section describes some of the methods we use to accommodate the routing application.

The \texttt{update\textunderscore status()} method inserts this node’s identifier, IP address, and timestamp in the distributed table. This method is called every time the routing application starts, but the information is only updated when the node enters the network for the first time. Its identifier and IP address can be set as a constant in a configuration file. Note that this information will be used to access the database application, so it is assumed that each node that runs the routing application, also runs the database application. The timestamp can be updated manually if a node changes its data, by calling a maintenance method of this class (\texttt{update\textunderscore time()}).

The \texttt{get\textunderscore db\textunderscore clients()} method returns the relevant nodes based on the query parts it gets as a parameter. Using a TTL parameter in the configuration, it refreshes the list of nodes in the network after a certain amount of time. Because this information usually does not change that often, there’s no need to refresh it every time this method is called. The rest of this method implements the algorithm from Section 3.3.4, which is quite straightforward.

The last function noteworthy in this class is \texttt{update\textunderscore routing()}, which was already mentioned in the previous section. Its implementation is simple, but an important thing to note is that it sometimes needs to update information even if the list of nodes remains the same. Say for instance that a node did not have results for a particular query part. When this node gets updated, it will be queried, because new information might be present. If after the update, results turn out not to be presented for this query part, the list of nodes that has results for this query part is the same. We need to update the timestamp of that query part in Riak, so that it will be newer than the timestamp of the updated node.

All the aspects of the routing application that have been discussed up till now can be summarised in Figure 4.3.
CHAPTER 4. IMPLEMENTATION

4.3 Communication between nodes

In the previous sections it is already mentioned that the communication between a client and the routing application, and the routing application and the database application, uses TCP. TCP is capable of efficiently providing continuous streams of data, which is perfect for our needs. There was also an option to use libraries such as Netty (for Java) and Twisted (for Python) but since our application uses straightforward, low-level communication we preferred implementing our own protocol. The protocol is illustrated in Figure 4.4.

Messages that are sent between applications can have different lengths. Easiest way to make sure that an application has received the complete message is by either using a special termination value, or specify the length of the message in a fixed part of the message. Because the data that we send can have many different values (and might accidentally include a termination value), we decided to send the message length in the header of the message. The header is 4 bytes, but for the current needs using only 3 bytes is sufficient, that gives us a maximum message length of $2^{3*8} = 16,777,216$ bytes (16 MiB).

Each time a message is received, the first 4 bytes are always received, and based on the extracted message length, the rest of the message is received. In both Java and Python we implemented a Network class that provides the packing and unpacking of the messages, so that for the application only the actual contents is visible, which is the command and the (optional) data.

In the previous sections the commands were already discussed in terms of their meaning. To get a command from one application to another, it is represented by a constant integer, which is sent as a 1-byte value in the message, thus providing space for $2^8 = 256$ different commands. In the current implementation we only use three commands and a few return signals. These signals, like for example NO_RESULT, take the place of the command byte in the message when the database application sends a result back to the routing application, or when a routing application sends a
message back to the client.

The Protocol class in both applications keeps a list of commands and return values and has – with a few simple functions – the responsibility of checking a message for or create a message with a certain command or return value. In the application code, for instance in the RequestHandler class, the only thing you need is the right command or return variable name, and the Protocol and Network classes take care of the rest.

Extending the protocol with new commands or return values is an easy task, it is a matter of creating a new constant, and using that in the rest of the application. Because TCP is used as the underlying protocol, any programming language that supports it can communicate with our applications, as long as they stick to our 4-byte header, 1-byte command protocol.

4.4 Ingesting the data resources

All the main applications are now in place, but we still need one application that is responsible for ingesting data resources into the indexes. This application will be described here, however, we will only go into the most important details, as it is less important than the main applications.

4.4.1 Parsing

The application starts with parsing a given directory of files, using a parser from Jena. It parses several types of serialised Linked Data formats as described in Section 2.1.4 and is written in Java, which makes it easy to store the parsed triples directly into Lucene and Berkeley DB.

For each triple that is parsed, the object is checked. If it is a literal, then it is indexed in Lucene, otherwise in Berkeley DB. Before we actually index the triples, there is an option to write part of each triple to a file in plain text. This file can then be used as a list of queries for several testing procedures (see Chapter 5). If the triple is intended for Lucene, we split the object part of the triple into separate keywords. By obtaining a list of stop words using the getStopwordSet() method of Lucene’s EnglishAnalyzer object, we filter out unusable keywords. The rest is written to the file. For triples that are directed towards the Berkeley DB index, we simply write the predicate/object pairs to the file.

4.4.2 Indexing predicate/object pairs into Berkeley DB

Next, it is time to index the triples. With creating a Berkeley DB index, the DatabaseConfig defines some index-specific properties (this is mentioned briefly in Section 4.1.2) such as the underlying data structure (a B-tree), the page size (8192 bytes seem to work fine) and whether or not to allow sorted duplicates. This is set to true so that one key can have multiple values (URIs), in sorted order, which is exactly what we want. Note that the page and also the cache size can be tuned in very much detail, to get the optimal performance for the underlying operating system. This is however beyond our investigation. Finally, the actual data is stored using two Berkeley DB DatabaseEntry objects which hold the byte representation of the key and value we want to store.

4.4.3 Indexing literal objects into Lucene

Given the nature of the index – providing full-text search – the procedure to store information in Lucene is a bit more complex. Lucene uses Document objects with fields to index data. To store triples we use a StringField object that indexes and stores the URI of a triple, and a TextField object that tokenises and indexes the literal object data of the triple. Note that there is an important difference in indexing, storing and tokenising data in Lucene. Storing it makes sure you can retrieve it later and indexing makes sure you can sort on this field (we need both for subject URIs). Tokenising makes sure the data is split up into parts – tokens – optimising it for search. We don’t store TextField, as we don’t want to retrieve that data later.
Lucene documents with the same value in the `StringField` object, and thus the same URI, are not combined automatically. That’s understandable, for a full-text index they are just two documents that happen to have a field value in common. In our application, however, the consequence is that duplicate results will be returned. To prevent this from happening we keep the `Document` objects from each parsed input file in memory first. When we encounter a triple that has a subject URI for which the `Document` object is already in memory (it represents the same data source), we concatenate the literal object of that triple into the `TextField` object of the `Document` object. As soon as the input file is processed, we flush all the documents into the index.

Storing `Document` objects in Lucene is straight-forward. The only thing to note is we use the `StandardAnalyzer` to process the `TextField` objects. It uses a grammar-based tokeniser, normalises the tokens, converts them to lower case and removes the stop words, based on the English language. It is important to use the same analyser here that we use for querying (see Section 4.1.1), otherwise a query might for instance be tokenised in a different way and not match the indexed tokens.

All the applications in the system are now implemented. As indicated above, an overview of all the programming languages, compilers, libraries and their versions are listed in Appendix A.
Chapter 5

Performance Testing Setup

This chapter describes the tests that were designed for measuring the performance of the database and routing application, which addresses subproblem P3. The performance is measured in terms of query execution time, ping signal time, throughput, and number of connections that are made between nodes, to check the system for meeting requirements R7, R10, and R11. The following parameters are taken into account:

- Query type
- Query result size
- Query
- Node size
- Number of nodes in the network
- Geographical distribution of the network
- Number of nodes in the network that return results for a query
- Query routing strategy

It is clear that using all those variables will cause a ‘test space explosion’, therefore we decided to split the tests up into two sections: single node tests and network tests. In the single node tests we tested the database application and used the parameters ‘query type’, ‘query result size’, ‘query’, and ‘node size’. The network tests fixed those parameters and tested the routing application using the ‘number of nodes in the network’, ‘geographical distribution of the network’, ‘number of nodes in the network that return results for a query’, and ‘query routing strategy’. In the network tests we tried to fix the parameters that influence the database application to the extent possible, making the two tests independent of each other.

All tests were run as a test plan in JMeter, to simulate requests to the database and routing application. We used a cluster managed by OpenNebula cloud software, on which each node ran in a separate OpenNebula Virtual Machine (VM). JMeter also had a separate VM for simulating the requests. Details on the exact software and hardware that was used can be found in Appendix A.3.

In the next section we describe the data that we used for the tests. The single node tests are then described in Section 5.2 and details on the network tests are described in Section 5.3.

5.1 Data selection

An important decision for testing is what data and queries to use. In any performance test, one could use a randomly generated data collection, that is queried using randomly generated
queries, to make the tests as controllable as possible. In our system, that is built for a very specific purpose, we could generate data randomly using for example a data generator from the Linked Data Benchmark Council \[13\]. Less ideal is generating queries with specific result set sizes (discussed in Sections 5.2.1 and 5.3.1).

We decided to use real data: the data dumps from two collections that are very representative for the collections that will be used in a production environment: the British National Bibliography collection\[1\] consisting of ±100 million RDF triples, and the British Museum Collections Online\[2\] consisting of ±200 million RDF triples.

### 5.2 Single node tests

In the single node tests, the focus is on the performance of the database application, measured in query execution time and throughput. Each of those performance measures were done in separate tests. Testing the query execution time is discussed in Section 5.2.3, the throughput in Section 5.2.4. First however, we discuss the parameter values in Section 5.2.1 and the data preparation in Section 5.2.2.

#### 5.2.1 Parameters

Because both types of query parts are processed by different parts of the database application, we wanted to test both types separately and in combination with each other. We therefore used the following query types in the tests:

- Keyword only
- Keyword and predicate/object pair
- Predicate/object pair only
- Two predicate/object pairs

For each of these types we could have just used random queries (as discussed in Section 5.1), but we wanted to make sure that we tested for different sizes of result sets, so that we don’t test queries that only return one result or even no results at all. Note that a query with a keyword and two predicate/object pairs is not tested, it turned out to be too difficult to get the correct result sizes. We used the following result sizes:

- No results
- One result
- About 300 results

According to assumption A2 a collection of data resources will contain up to 25 million RDF triples. Given the size of the investigated collections and the potential future collection sizes in a production environment, we tested nodes with a size of 10, 20, 40, 60 and 80 million RDF triples. For the current requirements we then have a worst-case test scenario.

#### 5.2.2 Data preparation

To get the correct amount of results for each query type, the data collections introduced in Section 5.1 needed to be prepared. For each collection, we took the following steps.

First, we selected 10 million triples randomly from the collection, and ingested those into the database application. We then used the files with query parts (see Section 4.4.1) to help us find

---

1 Dated January 2015 – [http://www.bl.uk/bibliographic/download.html](http://www.bl.uk/bibliographic/download.html)
the query parts and combinations of query parts that returned the right amount of results. We
saved the files with query parts for the load test.
Next, in the rest of the collection, we removed all triples that could lead to results for the
chosen query parts (using basic command line tools like `grep` and `sed`). The 20, 40, 60 and 80
million triples collections then returned the same amount of results for those query parts. It is
then fair to compare measurements between different node sizes, because we can be sure that
differences in measurements are not because of a different amount of results that was returned.
Finally, we split the rest of the collection up to obtain 20, 40, 60 and 80 million triples collec-
tions.

5.2.3 Single node query execution test
For this test we used one query for each result size, and each type of query, thus, 12 queries in
total. We also used one extra query of the type keyword and predicate/object pair to warm-up the
application (see Section 4.1.4). The queries used for this test are listed in detail in Appendix B.
Next, we set the cache of Berkeley DB to a minimum, which is 20 KB (it needs some cache to
keep a working set for traversing through the B-tree). For Lucene there is no cache setting, or at
least it is not configurable. The database application is compiled as an executable Java archive
file, which we ran with the `-Xmx2500M` option to give the Java virtual machine a maximum heap
size of 2500 MB. Finally, we ran the tests with the following steps:

1. Start database application
2. Run warm-up query
3. Run each of the 12 queries, with one second pause in between
4. Stop database application
5. Run `sudo sync && echo 3 | sudo tee /proc/sys/vm/drop_caches`

For both collections, and each of the 5 node sizes, we ran the above steps 5 times, to be able
to take an average of 5 execution time measurements. Note that step 5 is necessary to clear the
operating system caches as much as possible. We also tried to do a clean restart of the operating
system, but that appeared to have no extra effect over just clearing the caches.

5.2.4 Single node load test
For the load test we wanted to measure the maximum throughput of the database application, so
setting the parameters as strictly as in the previous test wasn’t necessary (or even feasible).
To have enough queries to be executed we used the files that we saved as described in Section 5.2.2;
one file with keywords, another file with predicate/object pairs. We let JMeter generate
a query on each request, by picking a query part from both files (more on JMeter in Section A.3.1).
This results in a query with both a keyword and predicate/object pair each time, and although
both indexes in the application are set to work, the intersection of the intermediate results usually
ends up in having no final result, which is obviously unrealistic. We therefore decided to add a
random variable to this process that in – on average – one out of four requests, only generates
either a keyword or a predicate/object pair as query (which one is in turn determined by another
random variable). That way the number of results will be more realistic, and the application’s
indexes are both used in most of the requests. Considering the number of tests to perform we were
unable to test for different random variable settings. The test was run with the following steps:

1. Start database application
2. Run the desired number of queries per second for 5 minutes
3. Stop database application
4. Run `sudo sync && echo 3 | sudo tee /proc/sys/vm/drop_caches`

We ran the above steps 6 times, starting from 100 queries per second to 600 queries per second (with 100 queries per second increments), using only the 80 million triples dataset from the British National Bibliography as prepared in 5.2.2 to have a worst-case scenario.

### 5.3 Network tests

In the network tests we are not interested in absolute performance figures, but rather in the relative difference between performance when we increase the network size or geographical distribution, and switch the query routing strategy from flooding to routing.

The parameter values are described in Section 5.3.1 and the data preparation in Section 5.3.2. Section 5.3.3 describes the network query execution test and Section 5.3.4 the network load test.

#### 5.3.1 Parameters

We fixed the node size to 40 million RDF triples, which is more than the 25 million triples stated in assumption A2 but not too difficult to manage (given the number of nodes that are needed).

The network sizes used are 5 nodes, 10 nodes, and 20 nodes. For each of these sizes we measured the performance when the query returned results from:

- No nodes
- One node
- 30% of the nodes in the network

We split the ‘query routing strategy’ parameter up into three settings:

- Flooding: Riak is not used
- Routing initialisation: routing is enabled but no routing information exists in Riak at the start of the test
- Routing: the routing information is already present in Riak at the start of the test and can be used during the test

To simulate a more realistic environment we wanted to distribute the network over multiple geographical locations. However, as we were restricted to using only one data centre, we simulated this parameter by implementing a latency on each connection being made between nodes. We used a latency of 30ms, which is average for regional round trips within Europe\(^3\) and therefore covers latencies within the United Kingdom, where a production network would reside.

#### 5.3.2 Data preparation

In contrast to the single node tests, the data preparations for both network tests are quite different.

For the network query execution test we used the British National Bibliography collection, with query 6 from Appendix B.2. We created 7 different datasets, one that has no results for the query, and 6 that have each a different set of results (of 50 URIs each) for that query. We need 6 sets that return results because 30% of the maximum network size (20 nodes) should be able to return results, and one set that returns no results for all other nodes. Note that for a fair comparison of measurements it would have been better to have a constant number of total results, no matter the amount of nodes that returned results, but this was too difficult to achieve. The datasets were constructed randomly from the entire collection, and like the data preparations for

\(^3\)http://www.verizonenterprise.com/about/network/latency/
the single node tests, tools like grep and sed were used to remove triples that could lead to more results.

For the network load test we split the 80 million triples dataset from the British National Bibliography collection used in Section 5.2.4 into 20 parts of 4 million triples each. To make the node sizes more realistic we added random triples from the British Museum Collections Online collection to get a node size of 40 million triples. Note that by adding data this way the nodes are not unique (the British Museum Collections Online collection consists of only 200 million triples) but we thought it was more important to use representative data and node sizes.

5.3.3 Network query execution test

For this test we used the same settings for the database application as in Section 5.2.3. For Riak we set the ring size according to the size of the network, using the recommended settings from the documentation: 64 for a 5-node network, 128 for a 10-node network, and 256 for a 20-node network. As mentioned in the previous section, we ran the test with query 6 from Appendix B.2, using the following steps:

1. Start the routing applications
2. Run warm-up query
3. Run query, flush routing information afterwards if needed, repeat 5 times
4. Stop routing applications

After each batch, the settings in the routing application (routing or flooding, with or without latency), the network size and the number of nodes that return results were changed to obtain measurements with all possible parameter settings. Note that due to time constraints the database application was not restarted after execution of each query.

5.3.4 Network load test

In the network load test we wanted to test the different parameter settings under a specific load. Since the network tests were designed to be independent from the single node tests – and we therefore can’t compare absolute query execution time or throughput measures – we use this test solely for comparing relative query execution times, ping times, and network load measures when changing the routing and latency parameters.

We used the same idea to create queries as with the single node load test (see Section 5.2.4), but to be able to compare the different parameter settings better, we used the same random feed with each test. The network size was set to 20 nodes (and therefore Riak’s ring size to 256), and we set the target load to 100 queries per second. The test was run with the following steps:

1. Start database applications
2. Start routing applications
3. Run 100 queries per second for 5 minutes
4. Stop routing applications
5. Stop database applications
6. Flush Riak
7. Run sudo sync && echo 3 | sudo tee /proc/sys/vm/drop_caches
After each batch, the settings in the routing application (routing or flooding, with or without latency) were changed. Besides the measurements that were recorded with JMeter, we also used logging information directly from the routing application to extract the number of connections that were made between nodes. Note that since we only had to run this test 6 times, it was – in contrast to the previous section – more feasible to clear the caches between each batch.
Chapter 6

Test Results

In Chapter 5, tests were designed to measure the performance of the database and routing application. In this chapter, we will look at the results and try to determine how well those applications performed. The tests were split up into single node tests – for which the results are discussed in Section 6.1 – and network tests – discussed in Section 6.2. Recall that the queries from the British National Bibliography collection are listed in Appendix B.2 and from the British Museum Collections Online in Appendix B.1. A summary of the test results is given in Section 6.3.

6.1 Single node tests

This section describes the single node tests, divided into two parts, the query test in Section 6.1.1 and the load test in Section 6.1.2.

6.1.1 Single node query execution test

In this section, all figures show – for a particular collection size – five individual runs and the average calculated over those runs. This average excludes outliers, defined as the measurements that fall more than 1.5 inter quartile ranges outside of the first and third quartile of the measurements. The reason that the five individual runs are included as well, is that sometimes an odd measurement is not eliminated by the above method, and then clutters the average. By showing the separate runs you get a better idea of the performance.

Each run has a separate measurement for the query time for each index (the \texttt{query()} method described in Sections 4.1.1 and 4.1.2), the fetch time (the \texttt{hasNext()} and \texttt{next()} methods described in Section 4.1.3), and the overhead, which is all other work that needs to be done when handling a request. This includes receiving and parsing commands, parsing the query, and sending results back to the routing application.

Query execution time in relation to data collection size

Perhaps most important in this test is whether the query execution time increases significantly when we increase the collection size. Figure 6.1 visualises queries that have both a keyword and one predicate/object pair, and return no results (Figure 6.1a) and one result (Figure 6.1b) respectively. It appears that for a collection size up to 80 million RDF triples the execution time remains fairly constant. It looks like extra overhead is involved in query 4 on the larger collections, but if we look at the results from other queries in this section, we will see that the overhead is unrelated to the collection size, and occurs at random. This is probably related to the resources that are shared in the VM environment.

Note that query 5 takes less time to execute, while it returns more results than query 4. This is because the merge-join algorithm has to fetch fewer intermediate results in query 5, as will be
discussed later. First however, we look at how the number of results that are fetched, influence the query execution time.

**Fetching results directly from one of the indexes**

The fetch time can consist of both executing the merge-join algorithm and – as part of that – fetching results from the indexes, or merely fetching results directly from one of the indexes. Let’s look at the latter first, by visualising the execution times of keyword-only queries 1, 2 and 3 for in Figure 6.2a, having no results, one result and 858 results respectively. If we compare this with predicate/object-only queries that have a similar amount of results, queries 7, 8, and 9 in Figure 6.2b, we see that Berkeley DB is much faster than Lucene. This is quite obvious as Lucene is a full-text index, providing richer search features – at the cost of performance.

The performance of Berkeley DB is even more clear in Figure 6.3, which shows the performance of query 12, with two predicate/object pairs. In query 12, the index goes through almost 100000 results, and the merge-join algorithm is used to merge the results from the two query parts. Results from the other type of query with two parts, a keyword and a predicate/object pair, are discussed next.

**Fetching results from both indexes**

Figure 6.4 again shows the results for queries 4 and 5, but now in combination with query 6. Query 5 takes less time than query 4 to execute, but as already mentioned, the intermediate results differ. In query 5 Lucene only has to go through half of the results compared to query 4, which halves the total execution time. Thus, although Lucene has twice the amount of results and it takes a little more time to process the `query()` method call, fetching the results is what actually takes up most of the time.

Query 6 takes quite an amount of time to execute, Lucene has to go through almost 4000 results, and Berkeley DB through over 100000 results. Comparing that with query 12 in Figure 6.3, we can see that fetching results from Lucene clearly takes up the majority of the execution time.
CHAPTER 6. TEST RESULTS

Figure 6.2: Execution time of queries from the British Museum Collections Online – collection size of 80 million RDF triples only

(a) Query 1, 2, and 3

(b) Query 7, 8, and 9

Figure 6.3: Execution time of query 12 from the British Museum Collections Online

Figure 6.4: Execution time of queries 4, 5 and 6 from the British Museum Collections Online – collection size of 80 million RDF triples only
CHAPTER 6. TEST RESULTS

Using a different data collection

The single node query test was performed with two separate data collections, with the goal of filtering out any collection-specific features that could affect the results.

We can see from queries 6 and 12 in Figures 6.5a and 6.5b respectively, that the results are similar. The collection size – for the sizes that we use – does not influence the performance. The absolute performance of the queries looks worse than with the other data collection, but this can be explained by the fact that in both queries 6 and 12, Berkeley DB has to go through 3 times the number of results compared to queries 6 and 12 from the previous collection. So we can see here that the number of results that are fetched has a linear relation to the execution time, which is a quite obvious result.

The figures with query execution times for all the queries that were tested – including the ones not shown here – can be found in Appendix C.1.

6.1.2 Single node load test

The load test results are visualised in Figure 6.6. Figure 6.6a shows the throughput that we configured in JMeter compared to the actual number of queries that the database application was able to handle. It is immediately clear that a target throughput of over 400 queries per second results in performance drops. By monitoring the VM we could see that the CPUs maxed out on loads over 400 queries per second.

Figure 6.6b shows the average query execution time for the same test. These measurements were averaged over 10 seconds, to make the figure more readable. The same conclusion can be drawn from this figure as from the previous figure, they are obviously related. A higher (or more unstable) average query execution time results in drops of throughput. This is even more visible with a target throughput of over 400 queries per second, at around 150 seconds. A clear peak in execution time in this figure, is visible as a drop of throughput in Figure 6.6a.

Yet another way to make the performance limit visible is looking at the ping times, which are – averaged over 3 seconds – depicted by Figure 6.7. We can see that with a target throughput of 500 queries per second, the database application starts to become less responsive. An overview of all the ping time figures can be found in Appendix C.2.

Figure 6.5: Execution time of queries from the British National Bibliography collection
CHAPTER 6. TEST RESULTS

Figure 6.6: Single node load test – results for a target throughput

Figure 6.7: Single node load test – average ping time for a target throughput
6.2 Network tests

After the single node tests, we measured the performance over different parameter settings for the network. The results of the network query test are described in Section 6.2.1, the results of the network load test in Section 6.2.2.

6.2.1 Network query execution test

The results of the query test are summarised in Figure 6.8. Here, only the averages over the 5 runs for each parameter setting are shown, using the same outlier elimination process as mentioned in the single node query test. It is also important to note that – as discussed in Section 5.3 – we are not interested in the absolute execution time. It’s clear that they are quite low, and although this is partly because only a maximum of 300 results are returned (in case 30% of the network returns results), it is mainly due to the database application using its caches. By doing so we will not have much variance in query execution time in the database applications, and we can focus entirely on the routing application and the effect that different parameter settings have on it.

Each average measurement is split up into the Riak lookup time (the `get_db_clients()` method described in Section 4.2.5), the database query time, the Riak save time (the `update_routing()` method in Section 4.2.5), and the overhead. The database query time consists of creating threads for each node that must be queried, creating `DBClientConnection` objects and the actual querying of each node, including joining the results. The overhead is – similar to the single node query tests – all other work that needs to be done when handling a request in the routing application, like receiving and parsing commands, parsing the query, and sending results back to the client library (in our case JMeter).

Flooding and the routing initialisation phase

We will look at flooding first. If there is no latency involved (Figure 6.8a), an increase in network size appears to have a linear effect on the execution time. Although the nodes are queried in parallel, the database application that takes the most time to process a query, determines the overall execution time in the routing application, and with 20 nodes in the network, there is just more chance of one node being slow.

If there is latency involved (Figure 6.8b), the effect of the network size tends more towards sub-linear, apparently what we saw in the non-latency test were (static) initialisation effects.

We can also see that it matters whether a query has results in the network, but how many results don’t seem to affect the query time that much. It is hard to say if this is caused by the database application’s cache, or by some initialisation that is happening in the routing application.

The measurements of the routing initialisation phase look quite similar to those of flooding – obviously queries are flooded, and on top of that information has to be retrieved from and saved to Riak. This can be seen as the worst case situation. We need to do a lookup in Riak to find routing information (which does not exist), and then save it as soon as all the nodes are queried.

Routing

When we can make full use of the routing information, things get interesting. Most noticeable is that queries for which no results exist, are really fast. We still have the overhead of doing a lookup in Riak, but then we can skip the entire database query step, including the latency that is involved. When there is no latency, this effect is smaller, the overhead of the Riak lookup is then a significant part of the execution time.

For queries that have results we see that the advantage over flooding is – with latency – not that big. This can be explained by the fact that for these queries nodes have to be queried, and in terms of the query execution time, it does not really matter if that is one or 30% of the network, or the whole network (as with flooding), it can – as already discussed – be done in parallel. Furthermore, and perhaps more importantly, the nodes that are skipped with routing, are the ones that don’t return results, which are therefore the fastest to query. However, it is important to note here that
Looking at the different network sizes we see an advantage with routing, the network size hardly seems to influence the execution time. This is a result of the number of nodes that have to be queried: in a network of 20 nodes, a maximum of 6 will be queried (30%), which is almost the amount of nodes when flooding a network of 5 nodes. Most likely, if the network size were bigger than 20 nodes, the execution time would be affected by the network size, but at least the point at which that happens is postponed compared to flooding. Therefore routing is more scalable.

When comparing flooding and routing, it is important to realise that routing consists of both the initialisation phase and the actual routing. In a real-world situation these two would coincide, the ratio depending on how many new queries are processed by the network compared to the number of queries for which routing information already exist. The reason that there is no test for this situation is because it is very hard to setup properly.

As a final, more general remark, we can see that the overhead involved in the routing application is quite significant. It is not clear whether this is overhead of using Python, or actually processing that cannot be avoided.

6.2.2 Network load test

Just like in the previous test, in the network load test we are not looking at absolute measurements. However, it is important to note that in Figure 6.9 the execution times – which are averaged per second – vary widely, some even take several seconds to complete. Apparently some of the queries return many results in almost all the nodes, which leads to result sets of over 100000 URIs being sent over the network (for instance the predicate/object query \( \text{rdf:type, bibo:Book} \)). Retrieving all the results in one go is therefore a problem, one we did not have in the network query test, because there we controlled the amount of results that were returned in each test.

Furthermore, Python sometimes seemed to have difficulties during peak load moments. Its Global Interpreter Lock mechanism prevents threads being run on separate CPUs simultaneously, clearly not ideal for our routing application which handles many threads.

Despite all this, the system still appears to be able to handle a throughput of 100 queries per

---

Figure 6.8: Network query execution test

This only holds for a low-load situation as was the case in this test. When the network deals with a higher load, the nodes that are queried unnecessarily, do consume scarce resources.

Retrieving all the results in one go is therefore a problem, one we did not have in the network query test, because there we controlled the amount of results that were returned in each test.

Furthermore, Python sometimes seemed to have difficulties during peak load moments. Its Global Interpreter Lock mechanism prevents threads being run on separate CPUs simultaneously, clearly not ideal for our routing application which handles many threads.

Despite all this, the system still appears to be able to handle a throughput of 100 queries per

---

1 Here, ‘run’ does not include waiting for I/O
Average query time

In Figure 6.9, both the differences between running the tests with flooding and routing, and with and without latency are similar to those of the network query execution test. From the routing results we can see no clear advantage of avoiding latency when no nodes have to be queried. This is either because the averages are raised by the queries that have more results (and thus higher execution times) and we therefore don’t see the effects, or there are just not that many queries that have no results (and for which the database query step can thus be avoided).

A remarkable aspect of the results in Figure 6.9a is the performance drop in the routing initialisation phase, between 200 and 300 seconds of the test. As this is not visible during the test with latency – which uses the same queries – the only reasonable explanation is that the sharing of resources in the underlying VM infrastructure causes some trouble.

Average ping time and network load

Figures 6.10 and 6.11 show the ping times and network load, both averaged per second. Also in these figures the ‘glitch’ mentioned above is visible. The rest of the ping time results look fairly normal.

The network load measurements show the number of connections that were made between the nodes during query processing. Here we can clearly see what was discussed in the previous section: although not directly related to the query execution time, routing saves resources by not querying nodes unnecessarily. In an environment where the latency is higher, or the query execution times in the database application are worse, this can make a huge difference.
Figure 6.10: Network load test – average ping time for a target throughput of 100 queries per second, in a network of 20 nodes

Figure 6.11: Network load test – average network load for a target throughput of 100 queries per second, in a network of 20 nodes
6.3 Summary of the test results

From the test results of the database application we could see that the query execution time is not related to the collection sizes that we used (up to 80 million RDF triples). We found that Berkeley DB is much faster than Lucene, which is quite obvious as Lucene offers different (more complex) query capabilities compared to Berkeley DB. It is very clear that Lucene takes up most of the execution time if both indexes are used for executing a query. In general we found that fetching results is what takes up most of the time, and is linear to the execution time. With the hardware that we used, the database application was able to handle up to 400 queries per second.

By looking at the test results of the routing application, we found that – with both routing and flooding – the execution time has a sub-linear relation to the network size. By using routing, queries that have no results are very fast, especially when there is latency involved (which we then can skip). Queries that do have results, are not much faster but the advantage is that resources are saved. Routing also decreases the network scaling effects, because less nodes have to be queried. From the load test we could see, just as an indication, that a throughput of 100 queries per second was feasible. We concluded however that it is better to retrieve results in batches than all in one go, and that Python is probably not the best choice for this type of application. We could also clearly see that with routing significantly fewer connections to nodes were made, so this obviously saves resources.
Chapter 7

Conclusions

This chapter concludes this thesis. Here, we will summarise the main contributions of the project and the limitations of the outcomes, as well as some future work directions.

In Section 7.1, we look back at the problem statement and subproblems that were defined in Section 1.2 and summarise how our system solves those problems, taking into account the requirements that were set in Section 3.1. The limitations of the system are discussed in Section 7.2, which lead to a number of directions for future work. These are discussed in Section 7.3.

7.1 Contributions

In this project, we were tasked with designing, implementing, and testing a query engine, capable of querying multiple decentralised Linked Data data collections, in a fault-tolerant and scalable way. This section describes how the problem statement and subproblems defined in Section 1.2 are addressed, and whether the requirements listed in Section 3.1 are met.

Design and implementation

To solve the main problem we first addressed subproblem P1 by defining the structure of a query: a keyword or a predicate/object pair, or combinations of these ‘query parts’ combined as a conjunctive query. This part fulfilled requirements R1 to R4.

We then looked at subproblem P2. We designed and implemented the query engine using open source libraries combined with our own built components (in accordance with requirement R9). The result is a peer-to-peer network that consists of a routing application that routes queries to the right node in the network, and a database application that performs the actual search. Both applications run on every node in the network, so that they all act as individual APIs for the system as a whole, and making the system fault-tolerant (meeting requirement R6). In the peer-to-peer network, a node can easily be added to or removed from the network, so requirement R8 is met as well.

In the database application’s internals, two indexes are used to store the data collection. One index for the triples used in full-text search, and another index for those used for conjunctive search with predicate/object pairs. The most common serialisation formats of RDF are supported to ingest data into the indexes, therefore fulfilling R5.

Performance testing

Addressing subproblem P3, testing the performance of the system, was a rather complex task. There were many parameters to test, and the specific nature of the query engine made it difficult to get representative data and queries for the tests. We tried to solve this problem to the extent possible, by using data collections that are likely to be used in a production environment, and independently testing the database application and routing application with those collections.
CHAPTER 7. CONCLUSIONS

We found that a single node – holding a 80 million triples collection and using a standard non-SSD VM setup with 4 CPUs and 7.5 GB of internal memory – could handle up to 400 queries per second, while the query execution time remained in the order of seconds. This is in line with requirements $R10$ and $R11$. For sizes of up to 80 million RDF triples, the collection size did not seem to have much influence on the execution time. It was the number of results that are retrieved that had the most significant and linear effect.

With testing the network we found that routing had a limited advantage in terms of execution time when using small network sizes. However, the query execution time increased less when the network size was increased, therefore making the system more scalable, and meeting requirement $R7$. Taking into account the number of connections that are made between nodes, we saw that routing saved resources.

Ideally we wanted to test for the maximum throughput of the network as well, but the design of the tests and selection of data and queries made that very difficult. The network did seem to be able to handle 100 queries per second, though.

From the above summary we can conclude that we have a solution for the problem statement in Section 1.2, which meets the requirements from Section 3.1. However, there are a few limitations, which will be summarised in the next section.

7.2 Limitations

During testing, we ran into two problems. First and most important is that results for a query are all retrieved in one go. This becomes problematic when in the network a very generic query is executed, that returns results from many nodes or with large result sets per node (or, of course, both). To abstract from details and stay focused on the main problem, nothing was implemented to prevent this. It turned out the detail cannot be ignored in practice.

The second problem is that the routing application was implemented in Python, which appeared to be not the best choice for this kind of application. The way Python handles threads is not ideal and could have caused some performance issues in the routing application. We also found that the overhead in the routing application was quite significant. At this point is not clear if Python causes this, but it is likely.

7.3 Future work

This section concludes this chapter and therefore this thesis. Here we will look at future work that can be done to improve the system. It can already be used in a production environment, but the improvements in Section 7.3.1 will make the system perform better, especially in relation to requirements $R10$ and $R11$. These improvements should therefore have priority over the more research-related future work that is described in Section 7.3.2.

7.3.1 Production ready

To address the limitations summarised in Section 7.2 it is necessary to first port the routing application to a different programming language. If we use for example Java, we could solve the threading problem and perhaps cut some of the overhead, but also more efficiently integrate the routing application with the database application.

Addressing the problem of retrieving all the results in one go is more complex, but also more important. It will most definitely improve the performance of the network, obtaining lower query execution time and (therefore) higher throughput, because it is fair to assume that most library applications only need results for each query in at most the order of tens instead of hundreds or more.

This can easily be implemented by using **paged results**, which means that only, say, 10 results are returned after each **RESULT** command. Things start to get more complicated if we want those
pages of results in order of relevance. Although a full-text index supports this kind of ordering, a key-value index does not. So for predicate/object pairs, we have to determine which triples are more important than others, perhaps by using multiple indexes. Another challenge here is how to merge the results from different indexes if they are not sorted.

A way to solve this is to first fetch all first fetch all the (sorted) intermediate results including their relevance to the query internally in the database application, then merge-join those results and finally page them using the relevance. This can all be done when the \texttt{QUERY} command is sent, and then with each \texttt{RESULT} command, a new page is sent over the network. The downside is that due to the relevance requirement, all intermediate results need to be fetched before any result can be sent back to the client library.

On the level of the routing application, consider a network of 20 nodes in which each node has one result for a particular query. If we require paged results, and a page contains 10 results, this introduces the question of which nodes to prefer.

In short, these are improvements which look like they cost minimum effort to design and implement, but in fact have quite some challenging details. However, addressing these challenges will make the system more mature.

### 7.3.2 Research

There are several possibilities for further improvements. Different forms of caching could be investigated to prevent querying nodes over and over again with the same queries. Caches could be stored locally in the routing application, but also in the a DHT. We have to be careful though that it stays easy to add or remove collections from the network and that the system remains fault-tolerant, otherwise we might not meet all requirements any more.

It is interesting to look at how the routing information evolves over time. In our tests we did not run in to any problems, but obviously the tests were not run for days. Ideally we would only store the most important routing information, based on how many times it is used. A counter should be implemented to take care of that, and all the routing information under a certain threshold of that counter should be removed to keep the DHT from growing indefinitely.

Related to this is investigating the ratio between new queries are that are processed by the network – for which no routing information yet exists – compared to queries for which information does exist. This could be monitored on a system in a production environment, and based on that information routing strategies could be adapted (like routing only certain queries or even disable routing altogether).

Future work could be done in discovering \textit{hot spots}, particular pieces of collections that get queried very often. It is interesting to see if there is a relation between features of a collection and where hot spots occur. Solving the problem could be easily done by replicating peers that contain them, but it is also interesting to see how hot spots develop.

A different direction of future work could be done in finding more intelligent ways of routing. It is for example possible to use a collection’s ontology to find relevant nodes for a query, in case the query has a predicate/object part. If a certain predicate is not found in the routing information, all nodes could be searched, just as is currently done.
Appendix A

Software and hardware dependencies

This appendix gives some more detailed information on the software that is used by the implemented applications, in terms of libraries, programming languages, compilers and their versions. Section A.1 elaborates on the software used in the database application and the indexer, Section A.2 on the what is used in the routing application.

A.1 Database application

- Java$^\text{TM}$ SE Runtime Environment (build 1.7.0_71-b14)
- Java HotSpot$^\text{TM}$ 64-Bit Server VM (build 24.71-b01, mixed mode)
- Lucene 4.6.0
- Berkeley DB 12cR1 (12.1.6.0.20) NC
  This is not the Java Edition, but the C edition with Java bindings, which in our experience had much better performance. Without encryption.
- Apache Jena 2.11.0
- Google Json 2.2.4
- Apache Commons$^\text{TM}$ Configuration 2.0
  This library is used to load all configuration settings from one file. Uses dependencies Apache Commons$^\text{TM}$ BeanUtils 1.9.2 and Lang 3.3.2.
- Apache log4j$^\text{TM}$ 1.2.16
  This library is used to handle logging.

A.2 Routing application

- Python 2.7.3
- Riak 2.0.5
- Python client for Riak 2.2.0
  Uses dependency Riak Protocol Buffers Messages 2.0.0.16
A.3 Testing

In this section we describe the software and hardware that was used for testing.

A.3.1 JMeter

Apache JMeter™ (we used version 2.12) is a load testing tool written in Java that can be used to simulate heavy load on any type of server. It uses a sampler to make a request, and one can set up a thread group containing a number of threads that perform those requests.

We built our own sampler to connect JMeter to our database and routing application, and make QUERY, RESULT, and PING commands to simulate a database client application. JMeter can be configured to read data from a file, so we used that to provide the sampler with query parts, which it used to construct queries, if necessary based on random variables.

For the query execution tests, we used a very basic JMeter test plan for the convenience of executing the queries consecutively, as described in the previous sections. The measurements were not recorded in JMeter but directly in the database and routing application, to see exactly where the execution time is spent. For the load tests however, we used more features.

We configured JMeter with two thread groups. One thread group was used for making the actual query requests, having 400 threads, a ramp-up period of 60 seconds, and a constant throughput timer set to the desired number of requests per second. The other thread group was used for ping requests, which had just one thread and a constant throughput timer set to one request per second. With the ping requests we could see when the server started to become unresponsive. Both thread groups used a so-called listener to write sample time measurements to a file.

The JMeter test plans were run using the same Java version as mentioned in Appendix A.1.

A.3.2 OpenNebula

All tests were performed on a cluster running OpenNebula 4.12.0 for cluster and VM management. The cluster consists of 14 Dell PowerEdge R720 machines running OpenNebula/KVM Hypervisor, 6 Dell PowerEdge R320 machines running a Ceph Datastore, one Dell PowerEdge R320 machine running OpenNebula Monitor, and one PowerEdge R320 machine acting as a gateway filtering traffic from the internet.

The following VM settings were used for the tests:

- For all tests, JMeter was run using a non-SSD VM with Debian 7.4.6 amd64, 8 CPUs and 8 GB of memory
- The single node tests were run using a non-SSD VM with Debian 7.4.6 amd64, 4 CPUs and 7.5 GB of memory
- The network tests were run using 20 non-SSD VMs with Debian 7.4.6 amd64, 2 CPUs and 3.8 GB of memory

Note that running the tests in an VM environment may incur caches to be used by the VM software, it was however outside our control to disable those caches (if any were active at all). Also note that we used smaller VMs in the network test, this was because these were more likely to reflect a production environment.
Appendix B

Test Queries

This appendix lists the queries that are used in the query execution tests (Sections 5.2.3 and 5.3.3).

B.1 British Museum Collections Online

The following table lists the queries that were used from the British Museum Collections Online:

<table>
<thead>
<tr>
<th>No.</th>
<th>Keyword</th>
<th>Predicate/object pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SimeonVanderSteen</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Yoshimaru</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Paintings</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Knight</td>
<td>{ecrm:P2_has_type, <a href="http://collection.../id/thesauri/authority/1%7D">http://collection.../id/thesauri/authority/1}</a></td>
</tr>
<tr>
<td>5</td>
<td>Young</td>
<td>{ecrm:P50_has_current_keeper, <a href="http://collection.../id/thesauri/department/G%7D">http://collection.../id/thesauri/department/G}</a></td>
</tr>
<tr>
<td>6</td>
<td>Intermediary</td>
<td>{rdf:type, ecrm:E22_Man-Made_Object}</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>{ecrm:P10_falls_within, <a href="http://collection.../id/thesauri/x122414%7D">http://collection.../id/thesauri/x122414}</a></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>{skos:related, <a href="http://collection.../id/thesauri/x6550%7D">http://collection.../id/thesauri/x6550}</a></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>{ecrm:P70i_is_documented_in, <a href="http://collection.../id/bibliography/10121%7D">http://collection.../id/bibliography/10121}</a></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>{ecrm:P70i_is_documented_in, <a href="http://collection.../id/bibliography/457%7D">http://collection.../id/bibliography/457}</a></td>
</tr>
</tbody>
</table>

The following namespace prefixes are used in the queries:

ecrm: http://erlangen-crm.org/current/
skos: http://www.w3.org/2004/02/skos/core#
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
The following table lists all the results for each query, and for each index in the database application. It also shows, in the cursor columns, where the result fetching process stopped. As you can see, sometimes not all results had to be fetched, because one of the indexes ran out of results (in conjunctive queries), which causes the merge-join algorithm to stop.

<table>
<thead>
<tr>
<th>No.</th>
<th>Lucene results</th>
<th>Lucene cursor</th>
<th>Berkeley DB (1) results</th>
<th>Berkeley DB (1) cursor</th>
<th>Berkeley DB (2) results</th>
<th>Berkeley DB (2) cursor</th>
<th>Combined results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>858</td>
<td>858</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>858</td>
</tr>
<tr>
<td>4</td>
<td>929</td>
<td>893</td>
<td>4482</td>
<td>-</td>
<td>-</td>
<td>4482</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2002</td>
<td>493</td>
<td>4973</td>
<td>4973</td>
<td>-</td>
<td>4482</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3765</td>
<td>3756</td>
<td>105956</td>
<td>105956</td>
<td>-</td>
<td>105956</td>
<td>189</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>1005</td>
<td>1005</td>
<td>-</td>
<td>1005</td>
<td>1005</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>1086</td>
<td>1086</td>
<td>3371</td>
<td>3367</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>1130</td>
<td>1130</td>
<td>1189</td>
<td>1189</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>5721</td>
<td>5721</td>
<td>105348</td>
<td>88085</td>
<td>313</td>
</tr>
</tbody>
</table>
### B.2 British National Bibliography

The following table lists the queries that were used from the British National Bibliography collection:

<table>
<thead>
<tr>
<th>No.</th>
<th>Keyword</th>
<th>Predicate/object pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SimeonVanderSteen</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>GBA819583</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Israel</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cooking</td>
<td>(event:agent, <a href="http://bnb.data.bl.uk/id/agent/HMSO">http://bnb.data.bl.uk/id/agent/HMSO</a>)</td>
</tr>
<tr>
<td>5</td>
<td>Testing</td>
<td>(dcterms:subject, <a href="http://bnb.../id/concept/place/lcsh/UnitedStates">http://bnb.../id/concept/place/lcsh/UnitedStates</a>)</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>(event:place, <a href="http://bnb.data.bl.uk/id/place/Eindhoven">http://bnb.data.bl.uk/id/place/Eindhoven</a>)</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>(dcterms:subject, <a href="http://bnb.../id/concept/place/lcsh/Ireland">http://bnb.../id/concept/place/lcsh/Ireland</a>)</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>(event:place, <a href="http://bnb.data.bl.uk/id/place/Cheltenham">http://bnb.data.bl.uk/id/place/Cheltenham</a>)</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>(event:agent, <a href="http://bnb.data.bl.uk/id/agent/Prentice-Hall">http://bnb.data.bl.uk/id/agent/Prentice-Hall</a>)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(rdf:type, bibo:Book)</td>
</tr>
</tbody>
</table>

The following namespace prefixes are used in the queries:

- event: [http://purl.org/NET/c4dm/event.owl#](http://purl.org/NET/c4dm/event.owl#)
- rdf: [http://www.w3.org/1999/02/22-rdf-syntax-ns#](http://www.w3.org/1999/02/22-rdf-syntax-ns#)
APPENDIX B. TEST QUERIES

The following table lists all the results for each query, and for each index in the database application, just like the previous section.

<table>
<thead>
<tr>
<th>No.</th>
<th>Lucene results</th>
<th>Lucene cursor</th>
<th>Berkeley DB (1) results</th>
<th>Berkeley DB (1) cursor</th>
<th>Berkeley DB (2) results</th>
<th>Berkeley DB (2) cursor</th>
<th>Combined results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>981</td>
<td>981</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>981</td>
</tr>
<tr>
<td>4</td>
<td>820</td>
<td>785</td>
<td>4266</td>
<td>4266</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1013</td>
<td>1008</td>
<td>2630</td>
<td>2630</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3613</td>
<td>3467</td>
<td>295830</td>
<td>295830</td>
<td>-</td>
<td>-</td>
<td>328</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>1086</td>
<td>1086</td>
<td>-</td>
<td>-</td>
<td>1086</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>1029</td>
<td>1027</td>
<td>4150</td>
<td>4150</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>1011</td>
<td>1011</td>
<td>2247</td>
<td>2242</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>3281</td>
<td>3281</td>
<td>287400</td>
<td>287400</td>
<td>321</td>
</tr>
</tbody>
</table>
Appendix C

Test Results

This appendix lists some figures related to the single node tests that were not shown in Chapter 6.

C.1 Single node query execution test

Here we list the query execution time results for each query that was tested.

C.1.1 British Museum Collections Online

The queries to which these figures refer can be found in Appendix B.1.

Figure C.1: Query 1

Figure C.2: Query 2
APPENDIX C. TEST RESULTS

Figure C.3: Query 3

Figure C.4: Query 4

Figure C.5: Query 5

Figure C.6: Query 6

Figure C.7: Query 7

Figure C.8: Query 8
### C.1.2 British National Bibliography

The queries to which these figures refer can be found in Appendix B.2.
APPENDIX C. TEST RESULTS

Figure C.13: Query 1

Figure C.14: Query 2

Figure C.15: Query 3

Figure C.16: Query 4

Figure C.17: Query 5

Figure C.18: Query 6
C.2 Single node load test

This section lists all the average ping time figures for the load tests that were performed.

Figure C.25: Average ping time for a target throughput of 100 queries per second

Figure C.26: Average ping time for a target throughput of 200 queries per second

Figure C.27: Average ping time for a target throughput of 300 queries per second

Figure C.28: Average ping time for a target throughput of 400 queries per second
Figure C.29: Average ping time for a target throughput of 500 queries per second

Figure C.30: Average ping time for a target throughput of 600 queries per second
References


REFERENCES


REFERENCES


[59] Jürgen Umbrich, Claudio Gutierrez, Aidan Hogan, Marcel Karnstedt, and Josiane Xavier Pereira. Eight Fallacies when Querying the Web of Data. In 4th International Workshop on Data Engineering Meets the Semantic Web (DESWEB), 2013. 8

