UNIVERSITY OF SOUTHAMPTON

Querying the Web of Data with Low Latency: High Performance Distributed SPARQL Processing and Benchmarking

by

Xin Wang

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Web and Internet Science Group
Electronics and Computer Science
Faculty of Physical and Applied Sciences

April 2014

"You will never be happy if you continue to search for what happiness consists of. You will never live if you are looking for the meaning of life."

Albert Camus

UNIVERSITY OF SOUTHAMPTON

Abstract

Web and Internet Science Group
Electronics and Computer Science
Faculty of Physical and Applied Sciences

Doctor of Philosophy

by Xin Wang

The Web of Data extends the World Wide Web (WWW) in a way that applications can understand information and cooperate with humans on complex tasks. The basis of performing complex tasks is low latency queries over the Web of Data. The large scale and distributed nature of the Web of Data have negative impacts on several critical factors for efficient query processing, including fast data transmission between datasets, predictable data distribution and statistics that summarise and describe certain patterns in the data. Moreover, it is common on the Web of Data that the same resource is identified by multiple URIs. This phenomenon, named co-reference, potentially increases the complexity of query processing, and makes it even harder to obtain accurate statistics. With the aforementioned challenges, it is not clear whether it is possible to achieve efficient queries on the Web of Data on a large scale.

In this thesis, we explore techniques to improve the efficiency of querying the Web of Data on a large scale. More specifically, we investigate two typical scenarios on the Web of Data, which are: 1) the scenario in which all datasets provide detailed statistics that are possibly available on a large scale, and 2) the scenario in which co-reference is taken into account, and datasets' statistics are not reliable. For each scenario we explore existing and novel optimisation techniques that are tailored for querying the Web of Data, as well as well developed techniques with careful adjustments.

For the scenario with detailed statistics we provide a scheme that implements a statistics query optimisation approach that requires detailed statistics, and intensively exploits parallelism. We propose an efficient algorithm called Parallel Sub-query Identification

(Ψ) to increase the degree of parallelism. Ψ breaks a SPARQL query into sub-queries that can be processed in parallel while not increasing network traffic. We combine Ψ with dynamic programming to produce query plans with both minimum costs and a fair degree of parallelism. Furthermore, we develop a mechanism that maximally exploits bandwidth and computing power of datasets. For the scenario having co-reference and without reliable statistics we provide a scheme that implements a dynamic query optimisation approach that takes co-reference into account, and utilises runtime statistics to elevate query efficiency even further. We propose a model called Virtual Graph to transform a query and all its co-referent siblings into a single query with pre-defined bindings. Virtual Graph reduces the large number of outgoing and incoming requests that is required to process co-referent queries individually. Moreover, Virtual Graph enables query optimisers to find the optimal plan with respect to all co-referent queries as a whole. Ψ is used in this scheme as well but provides a higher degree of parallelism with the help of runtime statistics. A Minimum-Spanning-Tree-based algorithm is used in this scheme as a result of using runtime statistics. The same parallel execution mechanism used in the previous scenario is adopted here as well.

In order to examine the effectiveness of our schemes in practice, we deploy the above approaches in two distributed SPARQL engines, LHD-s and LHD-d respectively. Both engines are implemented using a popular Java-based platform for building Semantic Web applications. They can be used as either standalone applications or integrated into existing systems that require quick response of Linked Data queries.

We also propose a scalable and flexible benchmark, called Distributed SPARQL Evaluation Framework (DSEF), for evaluating optimisation approaches in the Web of Data. DSEF adopts a expandable virtual-machine-based structure and provides a set of efficient tools to help easily set up RDF networks of arbitrary sizes. We further investigate the proportion and distribution of co-reference in the real world, based on which DESF is able to simulate co-reference for given RDF datasets. DSEF bases its soundness in the usage of widely accepted assessment data and queries.

By comparing both LHD-s and LHD-d with existing approaches using DSEF, we provide evidence that neither existing statistics provided by datasets nor cost estimation methods, are sufficiently accurate. On the other hand, dynamic optimisation using runtime statistics together with carefully tuned parallelism are promising for significantly

reducing the latency of large scale queries on the Web of Data. We also demonstrate that Ψ and Virtual Graph algorithms significantly increase query efficiency for queries with or without co-reference.

In summary, the contributions of this these include: 1) proposing two schemes for improving query efficiency in two typical scenarios in the Web of Data; 2) providing implementations, named LHD-s and LHD-d, for the two schemes respectively; 3) proposing a scalable and flexible evaluation framework for distributed SPARQL engines called DSEF; and 4) showing evidence that runtime-statistics-based dynamic optimisation with parallelism are promising to reduce latency of Linked Data queries on a large scale.

Contents

List of Figures

List of Tables

List of Algorithms

Declaration of Authorship

I, Xin Wang, declare that this thesis titled, 'Querying the Web of Data: Distributed SPARQL Optimisation Techniques and Evaluation' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Acknowledgements

First of all, I would like to thank my supervisor Dr. Thanassis Tiropanis for his invaluable guidance, support and encouragement. I am very grateful for his consistent encouragement, as well as patience, which kept me in hope. His vigour to research and positive attitudes towards life and work will continue to stimulate me to make progress in my future career. Meanwhile, I also would like to thank another supervisor Prof. Hugh C. Davis for his great suggestions. All in all, this thesis would never have gone this far without them. Moreover, I want to thank my colleges in the Web and Internet Science Group of University of Southampton, both past and present, for their generous help through my Ph.D. study. I am indebted to Areeb Alowisheq, Dr. Tope Omitola and Dr. Ian Millard. I have enjoyed all the discussions with them, both academic and non-academic. My personal thanks are due to many friends in Southampton, Jiadi Yao, Mr. Anthony Robson, Mrs. Ela Robson and etc. They have easy and enrich my life in the last four years. Finally, I would like to thank my parents and grandparents for their unconditional love and support.

Abbreviations

BGP	**B**asic **G**raph **P**attern
BSBM	**B**erlin **S**PARQL **B**ench**m**ark
DBMS	**D**ata**b**ase **M**anagement **S**ystem
DSEF	**D**istributed **S**PARQL **E**valuation **F**ramework
HJ	**H**ash **J**oin
IDP	**I**terative **D**ynamic **P**rpgramming
LD	**L**inked **D**ata
LAN	**L**ocal **A**rea **N**etwork
MSJ	**M**erge **S**ort **J**oin
NLJ	**N**ested **L**oop **J**oin
Ψ	**P**arallel **S**ubgraph **I**dentification
QEP	**Q**uery **E**xecution **P**lan
QPS	**Q**uery **P**er **S**econd
RDF	**R**esource **D**escription **F**ramework
SPARQL	**S**PARQL **P**rotocol **a**nd **R**DF **Q**uery **L**anguage
VG	**V**irtual **G**raph
VM	**V**irtual **M**achine
VoID	**V**ocabulary **o**f **I**nterlinked **D**atasets

Chapter 1

Introduction

The World Wide Web (or simply the Web) plays an important role in providing information. It contains a huge amount of interlinked documents[1] the number of which is still increasing. With the help of various tools such as search engines (e.g. Google), people can easily access a significant amount of the information on the Web. However, the potential of the Web is far from being fully exploited since most of information on the Web is stored as documents, which are readable by humans but not understandable by machines[2]. A Web with machine-readable information (i.e. data) will enable collaboration between machines and humans, as well as sophisticated programmatic processing. A demonstrating example was given by Berners-Lee et al. [2001], describing a scenario in which a family carries out daily tasks with the assistance of software agents based on the information on the Web. As shown in Figure 1.1, Lucy's mother needed to see doctors. Looking for an appropriate doctor requires identifying candidates according to mother's prescription, and checking the doctors' availability. Instead of doing these by herself, Lucy instructed her application to perform the task. The application firstly queried mom's prescription that is available as machine-understandable data, then queried for appropriate doctors in the same manner. In the above example, the basis of the automation is that the machine can "understand" the information of Lucy's mother and the doctors.

[1] Here a "document" refers to a set of texts that are readable by humans, such as an article, or a web page. Later in the text we use "data" to refer to structured information that can be processed by machines.

[2] There are applications that are design to understand specific types of documents on the Web. However, generally speaking there is no application can understand an arbitrary document on the Web, since the Web is not initially designed for machine processing.

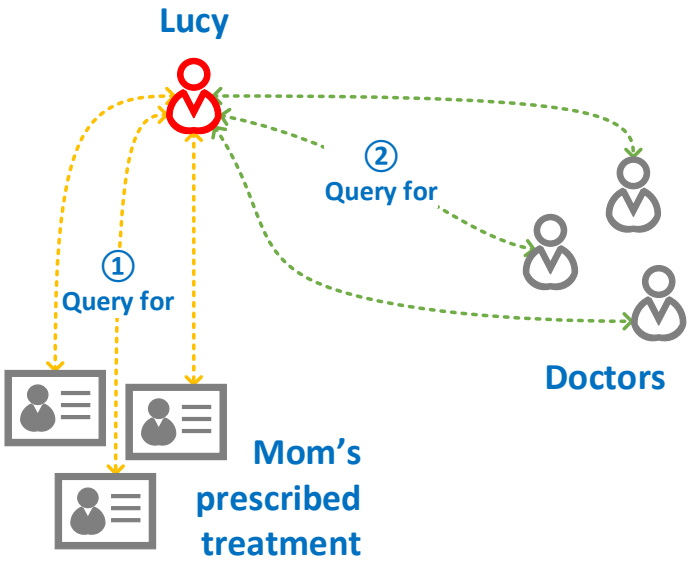


FIGURE 1.1: Booking a doctor using Semantic Web technologies

1.1 The Semantic Web, Linked Data and the Web of Data

To achieve machine readability, rather than solely relying on improving machines' natural language processing ability, researchers proposed the idea of associating web content with explicit semantics. This idea, called the Semantic Web, provides a collection of standard Semantic Web technologies that gradually transform the current web of documents into a collection of data. Data in the Semantic Web are represented using a common format called the Resource Description Framework (RDF) [Hayes and McBride, 2004] which is a graph-based data model. RDF is designed to be able to make assertions about both on-line documents and entities in the real world, which extends the expressibility of the Semantic Web. In addition, RDF data can be accessed and processed by any application that complies with standard Semantic Web technologies. Representing information using RDF makes the first step to transform the Web into a data space and thus enables automation on more complex tasks based on knowledge on the Web.

Machine-readable data is not the whole story. The Semantic Web is decentralised in nature (since it extends the WWW) and data on it can be isolated. This isolation reduces the interoperability of data and limits the usability of the whole Semantic Web. Consequently, when publishing RDF data, providing references or links to relevant datasets becomes equally significant. Guidelines have been introduced by Tim Berners-Lee for creating interlinked RDF data, referred to as Linked Data (LD) [Berners-Lee, 2006].

Repeatedly publishing LD will finally diminish data islands and lead to a global, interconnected data space, or the Web of Data.

1.2 Issues of Querying Linked Data

With the emergence of the Web of Data, both the quality and amount of LD are increasing. Small pieces of LD can be embedded in documents. In the meantime, it is also important to store relatively large LD in standalone repositories for convenient access and complex processing. These LD repositories can be accessed, or queried, using the SPARQL Protocol and RDF Query Language (SPARQL) [Prud'Hommeaux and Seaborne, 2008], which enables consumers of the Web of Data to send queries and receive results over HTTP connections.

In the Web of Data, it is very likely that applications access data (using SPARQL) from different repositories rather than a single source. Furthermore SPARQL queries tend to be much more structured and complex than the keyword-matching approach adopted by most of the contemporary searching engines. In the simple "finding doctor" example both the mother's prescriptions and the doctors' information can be stored at many different sites (Figure 1.1). To find an appropriate doctor the application has to issue multiple queries to those datasets, and these queries are related in a way that together they produce the desired answer. Extra care should be taken to dispatch these queries to datasets that probably can give part of the answer. Considering the large scale and distributed nature of LD, the ability of efficiently processing distributed SPARQL queries (in terms of time, network traffic etc.) can be a significant requirement of those applications. In the mean time, it is not clear whether such efficiency is achievable using standard Semantic Web technologies. For example, distributed queries is not explicitly supported in the first version of SPARQL specification, SPARQL 1.0. In the latest SPARQL 1.1 [Prud'hommeaux and Buil-Aranda, 2013], a *SERVICE* keyword is provided to evaluate certain triple patterns against explicitly provided data sources. The SERVICE keyword enables straightforward data federation when users know exactly which SPARQL endpoints can potentially answer a portion of a query. However, when a fair amount of data sources are involved, this method can be cumbersome and inefficient. In addition, SPARQL does not specify the infrastructure of distributed query execution,

where sophisticated techniques can take place and improve query performance as well as scalability.

From a broader view, distributed SPARQL queries share many characteristics with queries of distributed database management systems (DBMS). In a distributed DBMS, performance and scalability of query processing are closely related to network overhead and latency introduced in data transfer [Özsu and Valduriez, 1999], and the same relationship is also applicable in distributed SPARQL queries. As a result, many techniques that are developed to improve the performance of queries of distributed DBMS become relevant in the context of processing distributed SPARQL queries. For example, dynamic programming, which is widely used in distributed DBMS query optimisation, is also adopted in SPARQL query engines such as DARQ [Quilitz, 2008] and SPLENDID [Görlitz and Staab, 2011]. On the other hand, however, there are also differences between distributed SPARQL queries and distributed DBMS queries. For example, in most distributed DBMS data can be shipped among datasets. In addition, statistics of data, including frequency and distribution of certain patterns in the data, can be prepared according to the demand of query optimisation. However, the above advantages are not available in the LD cloud on a large scale. Furthermore, accurate and rich statistics, which are critical to query optimisation, are difficult to obtain on a large scale. There is a trend for datasets to adopt the Vocabulary of Interlinked Datasets (VoID) [Alexander et al., 2009] which provides an interoperative way to publish general information as well as statistics of the datasets. However, statistics provided by VoID are coarse and not flexible to meet different optimisation demands. Concerns of distributed DBMS, such as network traffic and query processing time, can be even more important in the LD context. When querying LD, all data are transferred through the Internet which is, generally speaking, slower and less stable than in distributed DBMS which tends to use local area network (LAN). The large scale of the LD cloud, and limitations such as network bandwidth and computing power of datasets bring more challenges into the research of distributed SPARQL query optimisation.

Moreover, in the LD cloud it is common to have multiple URIs referring to the same entity, which is known as co-reference. Co-reference exists in several fields such as linguistics and knowledge management, due to "inherently distributed and disparate nature of the information" [Glaser et al., 2007]. It is unlikely that a single URI can be accepted in all specific datasets in the LD cloud. In LD co-reference relationships

between URIs are represented using the *owl:sameAs* property [Carroll et al., 2012]. When querying the Web of Data, taking *owl:sameAs* statements into account increase the possibility of having additional results. Meanwhile, many unique challenges arise and extra care is required for processing distributed SPARQL queries having co-reference.

Similarities between distributed SPARQL and distributed DBMS make a strong case to exploit well-developed distributed DBMS techniques in the context of SPARQL. On the other hand unique challenges of distributed SPARQL processing demand specifically tailored approaches. Additionally, we hypothesise that efficiency in query processing cannot be achieved by an individual technique. Rather, it is the combination of techniques that leads to better performance. Due to the complexity of the Web of Data, it is unlikely to have one set of techniques that can provide desirable performance under all circumstances. To this end, schemes of techniques have to be tailored with respect to (w.r.t) certain environments.

1.3 Hypothesis and Contributions

In this thesis we explore a variety of techniques for improving the efficiency of distributed SPARQL query processing on large scale. It is posited in this thesis that, obtaining statistics from VoID files provided by RDF datasets, is the preferred or even only choice, for distributed SPARQL query processing in large-scale RDF networks. It follows that only limited statistics are available for query processing, and the presence of co-reference further decreases the accuracy of VoID statistics.

Based on the above assumptions we hypothesise that:

- VoID files or existing selectivity-based cost estimation methods are not sufficiently accurate for approximating QEPs that lead to minimum response time.
- It is possible to significantly reduce response time of LD queries on a large scale, by using both runtime statistics and parallelism.
- It is possible to address co-reference in LD queries within acceptable time, by considering co-referent URIs as a variable with multiple values.

The investigation of the aforementioned hypotheses would require: 1) typical scenarios, which could reflect the influence of statistics and co-reference on query efficiency, and in the mean time reasonably simulate the actual Web of Data; 2) an evaluation framework, which is capable of providing quantify and fair comparison among different approaches of LD queries in a wide range of scenarios on the Web of Data; 3) novel approaches designed to exploit statistics as well as co-reference in typical scenarios; 4) implements of the proposed approaches, which are compared with the state-of-the-art solutions over a large RDF network so as to obtain convincing evidence. Following this methodology, the contribution of this thesis could be summarise as follow.

We propose a benchmark called Distributed SPARQL Evaluation Framework (DSEF), that evaluates optimisation approaches in environments reflecting the actual Web of Data. By utilising artificial data and a virtual-machine-based infrastructure, DSEF is able to simulate RDF networks of arbitrary sizes. DSEF extends the data of the widely accepted Berlin SPARQL Benchmark (BSBM) [Bizer and Schultz, 2009] with co-reference, based on the distribution of co-reference in the real world. We also carefully modify the queries of BSBM in a way that emphases query efficiency in distributed settings while not loosing the original semantics of the queries. DSEF offers a set of scalable tools with which users can 1) generate an arbitrary number of triples following the BSBM data model; 2) generate co-reference for a dataset following the distribution of co-reference in the real world; 3) split and dispatch a dataset to remote endpoints following a given distribution; 4) generate detailed VoID statistics and 5) automatically evaluate distributed SPARQL engines and generate assessment reports.

We examine two typical scenarios on the Web of Data. In one scenario VoID files containing detailed statistics is provided by all datasets. We consider such statistics as the up bound of statistics that can be available on a large scale in the LD cloud. In the other scenario, co-reference is taken into account in query processing. As stated before, statistics of co-reference are unlikely to be included in VoID, and therefore more randomness will be introduced into query optimisation decisions. This scenario also covers a more general case that no reliable VoID statistics are available, which we consider as the lower bound of available statistics in the LD cloud. With respect to the characteristic of each scenario we investigate and propose different techniques. For the scenario with detailed VoID statistics, we propose an algorithm called Ψ to increase the degree of parallelism of query execution and thus better exploit available bandwidth and computing

resources. While parallelism can increase the efficiency of query execution, it tend to increase intermediate result size and thus increase network traffic, which is considered undesirable. Ψ identifies for a given query the components that can be processed in parallel without increasing network traffic. Since detailed statistics are available in this scenario, we adopt a static optimisation approach[3] that exhaustively searches for the optimal query execution plan. The static optimisation happens after Ψ breaks a query into smaller components which altogether are less complex to optimise than the original query. Furthermore, we develop a mechanism that maximally exploits bandwidth and computing power of datasets to further increase query efficiency.

For the scenario having co-reference, we propose a model called Virtual Graph (VG) that regards co-referent URIs as a variable with pre-defined bindings. Instead of issuing a separate query for each co-referent query of the original one, VG combines all co-referent queries into one query to save query requests and enable optimisation with respect to all co-referent queries. Due to the presence of co-reference, VoID statistics are no more reliable. As a result, we exploit runtime statistics in this scenario, and combine Ψ with a dynamic optimisation approach. A query is firstly processed by Ψ and each sub-query is optimised by the dynamic optimisation using runtime statistics. The break-then-optimise process repeats each time new statistics become available during execution. The parallel execution mechanism used in the previous scenario is applied here as well.

We implement both optimisation schemes as two distributed SPARQL engines, LHD-s ("s" for "**s**tatic optimisation") and LHD-d ("d" for "**d**ynamic optimisation"). Both engines are built using Jena[4], which is a well established Java-based platform for building Semantic Web applications. LHD-s and LHD-d can be used as standalone query engines or integrated into systems that require efficient LD queries. Using DESF we evaluate LHD-s, LHD-d and existing distributed SPARQL engines, and provide evidence that supports our hypotheses.

This thesis also contains contributions that are derived from the creation of LHD engines and DSEF, and are potentially beneficial for other researchers in the same area. A complete list of our contributions is detailed below.

[3] *Static optimisation* optimises queries *before* execution, as opposed to *dynamic optimisation*, which optimises queries *during* execution.

[4] `http://jena.apache.org/`

- We provide a comprehensive and in depth review of relevant distributed DBMS techniques and the-state-of-the-art approaches of distributed SPARQL processing.
- We propose and develop DSEF, a flexible and scalable benchmark for evaluating distributed SPARQL engines in environments reflecting the actual Web of Data. In addition we investigate the distribution of co-reference in the real world, based on which DSEF is able to simulate co-reference for arbitrary datasets.
- We propose an algorithm called Ψ that increases the degree of parallelism in query processing without increasing network traffic. We combine Ψ with a VoID-based static optimisation approach for the scenario in which detailed VoID statistics are available. Furthermore we propose a parallel query execution mechanism that exploits bandwidth of remote endpoints. Based on the above techniques we develop a distributed SPARQL engine called LHD-s.
- We propose a model called Virtual Graph that transfers a query and its co-reference into a single query with pre-defined bindings. We combine Virtual Graph, Ψ and a runtime-based dynamic optimisation approach for the scenario in which co-reference is taken into account. Based on the above techniques as well as the parallel execution mechanism we develop a distributed SPARQL engine called LHD-d.
- We design and perform experiments that examine LHD and existing engines in detail. The experiments confirm that our optimisation schemes substantially improve query efficiency. Furthermore, the experiments provide evidence that VoID files or existing cost estimation methods are not sufficiently accurate, and runtime-statistics-based query optimisation is promising.

1.4 Thesis Overview

The remaining part of this thesis is organised as follows:

Beginning with chapter 2 we explain terminologies used in this thesis and formally describe the basis of distributed SPARQL. Also in this chapter we describe the architecture

of distributed SPARQL processing. Following the preliminaries, query processing techniques that are well developed in distributed DBMS, as well as existing approaches of querying the Web of Data, are reviewed in chapter 3.

The core of this document includes the two LHD schemes and their evaluations. We firstly describe DSEF in chapter 4, to provide necessary details of later evaluations. In the following four chapters we provide details of the two LHD schemes and their evaluations. LHD-s is described in chapter 5, including: 1) the scenario that the scheme targets and corresponding challenges; 2) details of adopted techniques and implementation of the scheme. The evaluation of LHD-s is given in the following chapter (chapter 6), in which the scheme is thoroughly analysed. Following the same structure, LHD-d is described in chapter 7 and evaluated in chapter 8.

Finally, conclusions regarding techniques of the two schemes, open issues of querying the Web of Data, and our future plans are provided in chapter 9.

Chapter 2

Preliminaries

In thesis we base our discussion upon two concepts: RDF and SPARQL. On the Semantic Web information is represented using RDF, which is a World Wide Web Consortium (W3C) standard data model [Hayes and McBride, 2004]. RDF makes statements about web resources and real-world entities in the form of triples each of which consists of a subject, a predicate and an object. The subject refers to the resource described in the statement and the predicate expresses the relation between the subject and the object. RDF is very flexible to model various information on the Semantic Web.

SPARQL is the query language for RDF which is also a W3C standard. A SPARQL query contains triple patterns which are triples having variable subjects, predicates or objects. Each triple pattern matches one or more triples. Complex queries can be constructed by combining many triple patterns together. Since both RDF and SPARQL are W3C standards, they provide excellent interoperability across the Semantic Web.

In order to provide a clear context for discussions in this thesis, we introduce in this chapter basic terminologies and concepts that are necessary for the remaining part of this thesis. In the following we give formal definitions of RDF and SPARQL. Based on those definitions we describe the behaviours of distributed SPARQL query evaluation. In addition we describe four stages of distributed SPARQL processing that exist in many distributed SPARQL engines.

2.1 Basis of RDF and SPARQL Query Language

SPARQL is based on matching *graph patterns* against *RDF datasets*, each of which contains one or more *RDF graphs*. Intuitively, a *graph pattern* can be regarded as an *RDF graph* having variables. A successful matching between a *graph pattern* and an *RDF graph* represents the procedure that by replacing variables in the *graph pattern* with concrete values, the *graph pattern* becomes a sub-graph of the *RDF graph*. An example is given by figure 2.1. In this section we describe the aforementioned concepts using formal definitions, following RDF and SPARQL specifications (namely RDF Concepts and Abstract Syntax [Klyne et al., 2004], RDF Semantics [Hayes and McBride, 2004] and SPARQL 1.1 Query Language [Prud'hommeaux and Buil-Aranda, 2013]). For more details of RDF and SPARQL we refer the interested reader to the original documents and Angles and Gutierrez [2008], Gutierrez [2008], Pérez and Arenas [2009].

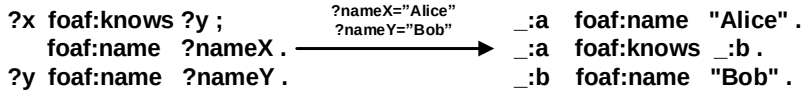


FIGURE 2.1: Matching a graph pattern (on the left) against an RDF graph (on the right).

We denote by I, B and L the pair-wise disjoint sets of URIs, Blank nodes and Literals respectively. An *RDF triple*, or simply a *triple*, denoted by (s, p, o), is a member of the set $(I \cup B) \times I \times (I \cup B \cup L)$. s is called the subject, p is called the predicate or property, and o is called the object. An *RDF graph* is a set of RDF triples.

Definition 2.1. An *RDF dataset* is a set of RDF graphs.

In our context an RDF dataset contains only one RDF graph for simplicity, since the number of graphs is irrelevant to the problem of this thesis. In the remaining part of this thesis we use RDF dataset and RDF graph interchangeably.

We denote by T the set $I \cup B \cup L$, which is the set of *RDF terms*, and by V the set of variables. A *triple pattern* is a member of the set $(T \cup V) \times (I \cup V) \times (T \cup V)$.

Definition 2.2. A *Basic Graph Pattern (BGP)* is a set of triple patterns.

In the SPARQL syntax a BGP is represented as a list of triple patterns enclosed by braces, as shown in table 2.1. BGPs are the building blocks of a more complex structure

called *graph pattern*. In this thesis we focus three basic forms of graph pattern shown in table 2.1: *conjunction graph pattern*, that returns results matching all presented BGPs; *alternative graph pattern*, that returns results matching either BGP; and *optional graph pattern*, that returns results matching the first BGP as well as results matching the second BGP if possible. We introduce binary operators Join, Union and LeftJoin to represent the aforementioned three BGP compositions. A graph pattern can be recursively defined as the following *SPARQL algebra expressions*:

Definition 2.3. 1) A BGP is a *graph pattern*; 2) if P_1, P_2 are graph patterns, then Join(P_1, P_2), Union(P_1, P_2) and LeftJoin(P_1, P_2) are *graph patterns*.

Table 2.1: Examples of SPARQL syntax and algebra

Query form	**SPARQL syntax**	**SPARQL algebra**
BGP	{ ?s :p1 ?v1 . ?s :p2 ?v2 }	BGP(?s :p1 ?v1 . ?s :p2 ?v2)
Conjuction	{ { ?s :p1 ?v1 } {?s :p2 ?v2 } }	Join(BGP(?s :p1 ?v1), BGP(?s :p2 ?v2))
Altenatrive	{ { ?s :p1 ?v1 } UNION {?s :p2 ?v2 } }	Union(BGP(?s :p1 ?v1), BGP(?s :p2 ?v2))
Optional	{ ?s :p1 ?v1 OPTIONAL {?s :p2 ?v2 } }	LeftJoin(BGP(?s :p1 ?v1), BGP(?s :p2 ?v2))

SPARQL is based around graph pattern matching, which is also the main subject of the research described in this thesis. Although SPARQL queries are not only composed by graph patterns, other components[1] are insignificant in the context of this thesis. In the remaining part we do not distinguish SPARQL queries and graph patterns. Matching a graph pattern against an RDF dataset is carried through matching triple patterns. The results of the matching are variable-value pairs having the following property: replacing the variables by their corresponding values makes the triple pattern a triple in the target RDF graph. A result of matching a triple pattern t against an RDF dataset d, is formally defined as a partial function $\mu : V \rightarrow T$, that $\mu(t) \in d$. $\mu(t)$ is the triple obtained by mapping the variables of t to RDF terms according to μ. μ is called a *solution mapping*. The domain of μ denoted by $dom(\mu)$ is the subset of V where μ is defined (in this case $dom(\mu)$ is the set of variables of t). Given a graph pattern p, we denote by $var(p)$ the set of variables occurring in p. The evaluation of p over an RDF dataset d, denoted

[1]For more details please refer to Prud'hommeaux and Buil-Aranda [2013]

by $eval(d, p)$, is a set of solution mappings $\{\mu | dom(\mu) = var(p) \wedge \mu(p) \in d\}$. In the remaining part of this thesis we refer the set of solution mapping as the bindings of a SPARQL query (the evaluation of a query over a dataset to be precise).

We say two solution mappings μ_1, μ_2 are compatible if for every $?v \in dom(\mu_1) \cap dom(\mu_2)$ it holds that $\mu_1(?v) = \mu_2(?v)$, i.e. any variable occurs in both mappings must be mapped to the same value. We denote the compatibility as $cmp(\mu_1, \mu_2)$ whose value is "true" if μ_1 and μ_2 are compatible. Let Ω_1, Ω_2 be sets of solution mappings, we define the join, union, and set minus of Ω_1, Ω_2 as the following equations:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge cmp(\mu_1, \mu_2)\} \tag{2.1}$$

$$\Omega_1 \cup \Omega_2 = \{\mu | \mu \in \Omega_1 \vee \mu \in \Omega_2\} \tag{2.2}$$

$$\Omega_1 \setminus \Omega_2 = \{\mu | \mu \in \Omega_1 \wedge \forall \mu'.\ \mu' \in \Omega_2 \rightarrow \neg cmp(\mu, \mu')\} \tag{2.3}$$

The results of evaluating conjunction, UNION and OPTIONAL queries are specified by the following equations:

$$eval(d, \text{Join}(p_1, p_2)) = eval(d, p_1) \bowtie eval(d, p_2) \tag{2.4}$$

$$eval(d, \text{Union}(p_1, p_2)) = eval(d, p_1) \cup eval(d, p_2) \tag{2.5}$$

$$eval(d, \text{LeftJoin}(p_1, p_2)) = eval(d, \text{Join}(p_1, p_2)) \cup (eval(d, p_1) \setminus eval(d, p_2)) \tag{2.6}$$

2.2 Basis of Distributed SPARQL Processing

By querying multiple RDF datasets, it is possible to get results that cannot be given by any individual dataset. We formally describe this behaviour and give the basic rules that are followed in this thesis.

Evaluating a SPARQL query over a set of RDF datasets is regarded as evaluating the query over the RDF graph that is the union of these RDF datasets[2]. We denote by D a set of RDF datasets, that is:

$$eval(D, p) = eval(G, p), \text{ where } G = \bigcup_{d \in D} d \tag{2.7}$$

In case of evaluating a query t_1 AND t_2, where t_1, t_2 are triple patterns, over two datasets d_1 and d_2, the following equation holds following equation 2.4 and 2.7:

$$\begin{aligned} & eval(d_1 \cup d_2, t_1 \text{ AND } t_2) \\ =& (eval(d_1, t_1) \cup eval(d_2, t_1)) \bowtie (eval(d_1, t_2) \cup eval(d_2, t_2)) && (2.8) \\ =& (eval(d_1, t_1) \bowtie eval(d_1, t_2)) \cup (eval(d_2, t_1) \bowtie eval(d_2, t_2)) \\ & \cup (eval(d_1, t_2) \bowtie eval(d_2, t_1)) \cup (eval(d_1, t_1) \bowtie eval(d_2, t_2)) && (2.9) \end{aligned}$$

For convenience, we call a join of results from the same dataset (e.g. $eval(d_1, t_1) \bowtie eval(d_1, t_2)$) a same-site join, and one of results from different datasets (e.g. $eval(d_1, t_1) \bowtie eval(d_2, t_2)$) a cross-site join. A same-site join equals evaluating all triple patterns together at a single dataset, which is the same behaviour of centralised SPARQL processing. Meanwhile, equation 2.9 shows that the results of a distributed query includes the results of all same-site joins and cross-site joins. Therefore, distributed SPARQL query processing is more than evaluating a query at different datasets and combining the results together. In addition, it is the cross-site joins that produce results which are not available at any individual dataset. As shown in figure 2.2, either D_1 or D_2 alone can answer the given query (i.e. the same-site join at either D_1 or D_2 is empty), while a result can be returned by evaluating t_1 and t_2 respectively at D_1 and D_2 (i.e. the cross-site join $eval(d_1, t_1) \bowtie eval(d_2, t_2)$ is not empty).

Some approaches [Quilitz, 2008, Schwarte et al., 2011] evaluate triple patterns as one query when possible (i.e. all cross-site joins of these triple patterns are empty) to reduce the number of results returned. However, it is unlikely to do so when many datasets

[2]The precise term here is a *merge* of RDF graphs, which is a little more complicated than union when *blank nodes* are contained in RDF graphs. Details of RDF merge can be found in Hayes and McBride [2004]

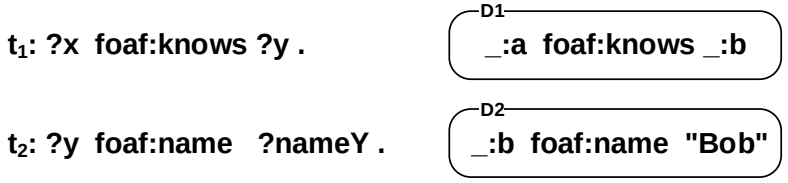


FIGURE 2.2: Two triple patterns t_1 and t_2 are evaluated against two datasets D_1 and D_2. Either dataset can produce results for the two triple patterns, while together give a valid result.

are involved[3]. In addition, evaluating multiple triple patterns together sometimes even increases the number of results [Quilitz, 2008]. Therefore, in our approaches (described in chapter 5 and 7) triple patterns are evaluated individually, and the results of each triple pattern are joined afterwards.

2.3 Stages of Distributed SPARQL Query Processing

In this section we describe a staged architecture that covers many distributed SPARQL query engines (e.g. DARQ, FedX, DSP, SPLENDID). This architecture divides distributed SPARQL query processing into a series of stages. Through these stages a SPARQL query string is transformed into an efficient execution strategy, called a query execution plan (QEP), composed by low level operations (e.g. executing triple patterns) on remote datasets. Figure 2.3 illustrates how distributed SPARQL queries are processed according to this architecture, in an environment that consists of a number of SPARQL endpoints.

Given a SPARQL query string it is firstly parsed into a SPARQL algebra expression that enable faster processing. Then the query engine loads service descriptions, which are built from the metadata of involved datasets. Service descriptions are crucial for later stages such as source selection and query optimisation. As stated in last section (section 2.2), each triple pattern is evaluated individually against all datasets in order to have complete results. In the meantime, it is possible to identify datasets that will not contribute any result for a specific triple pattern. This stage is called source selection that eliminates irrelevant datasets for triple patterns based on service descriptions.

[3]From equation 2.9 it follows that the more triple patterns and datasets, the less the chance that all cross-site joins are empty.

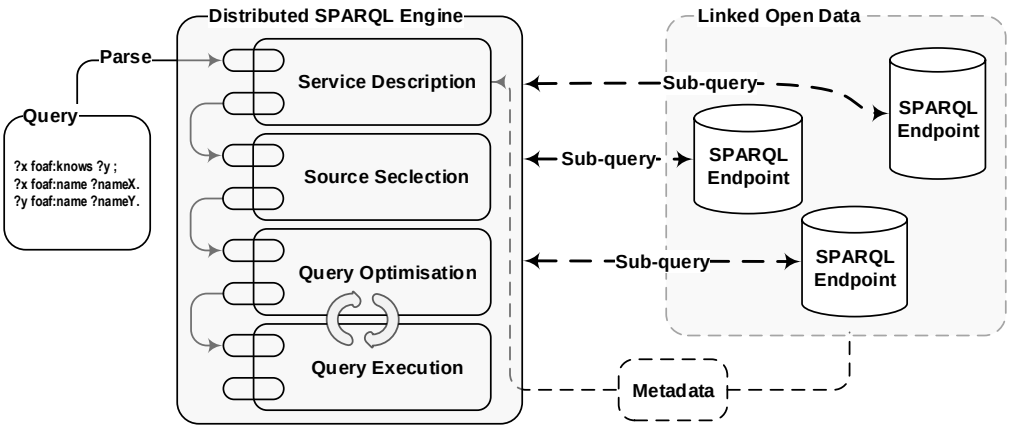


FIGURE 2.3: This figure shows different stages and relevant components of distributed SPARQL processing. Solid frames present local components while dashed frames present remote components. Local data flow is represented as solid lines while remote data transfer is represented as dashed lines.

After associating triple patterns with datasets, the algebra expression is transformed into a QEP that represents an execution strategy of the query containing operations of executing triple patterns and joining results. For a given query more than one QEPs can provide the same result but have different I/O, CPU and communication costs in terms of time. At the query optimisation stage, these equivalent QEPs are examined and the one with minimum cost is selected. The optimal QEP is executed at the query execution stage and the whole query processing finishes.

It is worth mentioning that the architecture described here is not the only possible way for processing LD queries. For example, another LD query strategy, called *link traversal based query execution* [Hartig, 2011], does not rely at all on metadata of data sources. Rather, approaches following the link traversal strategy (e.g. Hartig and Bizer [2009] and Ladwig and Tran [2011]) discover relevant data sources by resolving URIs in queries and intermediate results. In this thesis we focus on approaches that follow the architecture shown in figure 2.3 and others are out of the scope.

1. **Query parsing** compiles SPARQL query strings into algebra expressions, as shown in table 2.1. All later steps operate on the algebraic expressions rather than the original queries for the sake of efficiency and convenience. The query parsing algorithm is described in the SPARQL specification and implemented in

SPARQL platforms (e.g. Jena[4] and Sesame[5]). Distributed SPARQL engines usually leave the task of query parsing to the platforms on which they are built.

2. **Service descriptions** provide structural and statistical information of data sources for query processing. Service descriptions can be collected either from data sources directly or from third party indices of those data sources. Detailed and accurate service descriptions can lever the performance of later steps, especially for query optimisation. In the mean time, they cost a lot to obtain and thus compromise the scalability of linked data queries. In addition, it may be unrealistic to rely on detailed service descriptions on a large scale since issues of incompatible or incomplete service descriptions arise.

3. **Source selection** is the process by which each triple pattern in a query is associated to data sources that potentially can contribute result[6]. Source selection filters out irrelevant sources to reduces overheads of later stages. In addition it increases the accuracy of cost estimation of queries since only the statistics of relevant datasets would be considered. Relevant data sources are usually identified by analysing service descriptions, or directly checking the datasets for whether certain triple patterns can be answered. More details of these methods are described in section 3.2.

4. **Query optimisation** refers to the process that produces a QEP having the minimum cost w.r.t an objective cost function, called cost model. For a given query there are more than one valid QEP that differ in the order and implementations of operations. These QEPs compose the *search space* of query optimisation. The search space is examined by an *optimisation algorithm* which uses the cost model to assess each QEP and selects the one with minimum cost.

    - **Cost models** include cost functions to estimate the costs of QEPs based on statistics of service descriptions. The costs are usually measured in terms of execution time of I/O, CPU instruction and communication over the network. Communication is the dominant factor of distributed query processing, especially for LD queries which take place on the Web. The effectiveness of query

[4] `http://incubator.apache.org/jena/`

[5] `http://www.openrdf.org/`

[6] Source selection is close to the data localisation process [Özsu and Valduriez, 1999] in distributed DBMS that determines the fragments of data that are involved in a certain part of the query

optimisation is closely relevant to the accuracy of cost models. Inaccurate cost estimations lead to suboptimal QEPs, and thus affect the performance of the actual query execution. To be accurate detailed statistics, provided by service descriptions, are required. In the meantime collecting such service descriptions is costly. In practice it is a trade off between the detail of service descriptions and the accuracy of cost estimations.

- **Optimisation algorithms** explore the space of possible QEPs of given queries, and produce the optimal QEPs based on estimations given by the cost models. For queries whose search space is relatively small it is possible to identify the optimal QEPs using exhaustive search algorithms. For queries having large search space this approach becomes very time consuming. Instead, algorithms with less complexity, as well as heuristics are used for complex queries to produce QEPs close to optimal within acceptable time.

Depending on the timing when the QEP is constructed relative to the time of query execution, query optimisation can be divided into two categories: statistic optimisation and dynamic optimisation. Static optimisation prepares the QEP before query execution, based on estimations given by the cost model. Once the QEP is constructed, it can be reused in multiple query executions, and thus the cost of optimisation can be amortized. In addition, exhaustive search algorithms can be used since the cost of any candidate QEP can be estimated, and the optimal QEP can be found. Dynamic optimisation constructs the QEP during query execution. The accurate statistics of operations executed previously can be used to determine the best next move (the greedy approach). As a result, the optimal QEP may be missed. Dynamic optimisation does not rely on service descriptions as much as static optimisation does.

5. **Query execution** refers to the process of executing the QEPs produced in query optimisation. This stage determines the actual implementation of each operation, especially the implementation of join operations. In most distributed SPARQL engines, join operations are implemented following an iterator model [Graefe, 1993] which has been widely used in both centralised and distributed DBMS. Furthermore, the way in which those operations are executed is critical to the performance of query execution. For example, operations can be pipelined and/or executed in parallel [DeWitt and Gray, 1992]. More details are given in chapter 3.

Chapter 3

Related Work

Distributed SPARQL query processing can be regarded as a special case of query processing in distributed DBMSs. In Özsu and Valduriez [1999] distributed DBMSs have been classified w.r.t three dimensions: autonomy, distribution and heterogeneity. The autonomy refers to the distribution of control. The distribution dimension refers to the distribution of data. Two popular ways are client/server (CS) distribution, that data are hosted at server nodes while client nodes provide application environment; and peer-to-peer (P2P) distribution, that every node has full DBMS functionality and is able to communicate with each other. Heterogeneity refers to the differences among datasets, such as data models, query languages and networking protocols. Distributed SPARQL query processing involves query engines and independent SPARQL endpoints, which form a autonomous, CS-structured, homogeneous distributed DBMS whose scale is extremely large.

In the last a few decades many techniques have been developed to improve the efficiency of query processing in distributed DBMSs. These techniques inspire and stimulate development of distributed SPARQL query processing. However, most of these techniques are developed for distributed DBMSs that have smaller scale, more accurate statistics, and are often under uniformed administration. Meanwhile, detailed statistics of RDF datasets are difficult to obtain on a large scale, and the communication between query engines and SPARQL endpoints are via HTTP requests. These unique features of distributed SPARQL determine that it is not possible to use only distributed DBMS techniques, and set the demands for techniques specifically tailored for distributed SPARQL

processing.

This chapter is divided into three parts. First, we will give a comprehensive review of relevant distributed DBMS techniques, which serves as the technical background for distributed SPARQL approaches discussed in this thesis. This part focuses on techniques of query optimisation and parallelism. The characteristics of the LD cloud make it very difficult to build indices for RDF datasets on a large scale. The large number of RDF datasets raises a high bar for maintaining detailed indices or statistics. In addition, many RDF datasets can only be accessed via SPARQL endpoints which significantly limit the efficiency of gathering statistics for building indices. Rather, it is more promising to rely on metadata provide by RDF datasets using widely accepted format, such as VoID. As a result we consider that indexing techniques and data structures are more appropriate in environments of smaller scale rather than the LD cloud, and they are not covered in the review.

In the second part, we discuss how distributed DBMS techniques, as well as unique distributed SPARQL techniques, are adopted in existing distributed SPARQL engines.

Finally, a survey of RDF store benchmarks is given at the end of this chapter, which serves as the background of the evaluation framework that we propose in chapter 4.

3.1 Distributed Query Processing Techniques

Query processing in distributed DBMS involves a broad range of techniques including indexing data structures, query optimisation algorithms, cost models, implementation of joins and execution of query operations. In this section we focus on techniques significantly relevant to distributed SPARQL query processing, that mainly fall into query optimisation and execution.

Due to the importance of query optimisation in distributed DBMS, a large number of algorithms have been proposed. Existing optimisation algorithms belong to two basic categories: *exhaustive search* and *approximate algorithms*. With a sufficiently accurate cost model, exhaustive search algorithms guarantee to produce the optimal QEP at the cost of exponential time and space complexity. These algorithms generate all possible QEPs, estimate those QEPs using a given cost model, and identify the one with the

minimum estimated cost (i.e. enumerate all instances in the QEP searching space). A representative among exhaustive search algorithms is *dynamic programming*, which is first adopted in System R [Selinger et al., 1979] and later generalized for distributed DBMS in System R* [Lohman, 1988]. Dynamic programming produces the optimal QEP through several iterations. At the beginning it generates plans containing only one operation (access operations). In each iteration, it joins plans generated in earlier iterations to produce more complex plans (i.e. plans that contain more operations), and prunes those plans which have alternatives that can do at least the same work at a lower cost. At the end of the algorithm, the QEP which contains all operations and has the lowest cost is selected as the optimal one. More detailed descriptions of dynamic programming can be found in Kossmann and Stocker [2000].

The complexity of exhaustive search, including dynamic programming, makes it less favourable to optimise complex queries. To this end, approximate algorithms, which approximate the optimal plan with lower complexity, are proposed. Approximate algorithms used for query optimisation can be further divided into *heuristics* and *metaheuristics*. Heuristics adopts predefined rules to filter out QEPs, and thus reduces complexity. *Greedy algorithms* that adopt the "minimum cost rule" (i.e. keeping only the operation with the minimum cost in each iteration) are widely used. Between dynamic programming and greedy algorithms lies the family of *iterative dynamic programming* (IDP), which can be regarded as a combination of dynamic programming and greedy heuristics [Kossmann and Stocker, 2000]. Iterative dynamic programming provides the flexibility to balance between the quality of QEP and the complexity of query optimisation. Metaheuristics, including *randomized algorithms* and *genetic algorithms*, have not been applied in distributed SPARQL optimisation by the time this paper is written, and therefore are not discussed here. A detailed review of randomized algorithms and genetic algorithms in query optimisation is given by Steinbrunn et al. [1997].

The effectiveness of query optimisation is closely related to the accuracy of cost models. In DBMS, the cost of a QEP is usually estimated as the sum of the cost of all operations of this plan [Mackert, 1988]. The cost of an operation consists of various resource consumptions, such as CPU and memory, with different weight. In distributed DBMS, communication cost becomes significant and this is even more true in an environment like the LD cloud where communication is over the Internet via HTTP protocols. The cost of a QEP is closely related to the number of intermediate results. In statistic

optimisation, the number of intermediate results can be estimated using the concept of selectivity factor [Selinger et al., 1979, Poosala and Ioannidis, 1997], which is the expected fraction of results satisfying a condition. For example, the selectivity of having head in a coin tossing is about 0.5. In the context of this thesis, the selectivity factor of a triple pattern corresponds to the fraction of triples matching the pattern [Bernstein et al., 2007]. The estimation of the selectivity of triple patterns will be described later in this chapter. In dynamic optimisation, the accurate number of intermediate results can be obtained from previous execution. Rather than the resource consumptions, response time cost models focus on the time of query execution, and can reflect the advantages of using parallelism in query execution. These cost models are usually chosen when query response time is critical.

Query execution involves the implementations of the operations of a QEP and the way to execute those operations. In QEPs, join operation (denoted by $\bowtie$) is one of the fundamental and most difficult operations. Various implementations of the join operation, such as Nested Loop Join (NLJ), Hash Join (HJ) and Merge Sort Join (MSJ), have been proposed to improve the performance of query execution in DBMS. To execute $A \bowtie B$, NLJ scans for each tuple of A, all tuples of B to find the join results; HJ maintains a hash table to produce the join results; MSJ sorts the tuples of A and B and then alternately scans A and B only once to compute the join results. A detailed review of these join implementations is given by Mishra and Eich [1992]. In distributed DBMS, two extensions are proposed in order to reduce communication cost. One is Semijoin [Bernstein et al., 1981]. Given two tables A and B at two sites S_1 and S_2 receptively, Semijoin sends the column(s) of joined key(s) of A to S_2, computes the join of B with those A tuples at S_2, sends the join results back to S_1 and matches these results with A. This process can be formulated as $A \bowtie B = A \bowtie (B \ltimes \pi(A))$, in which $\ltimes$ is the semijoin operator[1], and $\pi(A)$ selects the join columns of A. Another variation is Bind Join [Haas et al., 1997] that simulates a Nested-Loops Join in a heterogeneous environment. Bind Join (BJ) is designed to take advantage of the fact that many component databases accept input parameters and therefore intermediate bindings are sent together with queries as additional filters. Both Semijoin and Bind Join send out extra data to filter out undesired results at the remote dataset and thus can reduce network overheads. In the meantime they also increase the number of operations [Özsu and

[1]The semijoin operator denoted by $\ltimes$ is different from Semijoin introduced above.

Valduriez, 1999]. Therefore, extra care should be taken to make sure that the network traffic caused by sending out filter data and additional operations is not beyond the amount of traffic saved by filtering out undesired result. Sometimes parallelism is an efficient technique that can reduce response time. To achieve parallelism, query execution is usually broken down into tasks that can be executed in parallel, and multiple threads are used to execute these tasks. For example, using multiple threads more than sites can be queried simultaneously rather than sequentially. We can also partition $A \bowtie B$ into $(A_1 \bowtie B) \cup (A_2 \bowtie B)$ where $A = A_1 \cup A_2$, and execute $A_1 \bowtie B$ and $A_2 \bowtie B$ in parallel. In a network in which the communication between data site is bursty, a join process can be "blocked" when waiting for the delivery of tuples. In such a case, Double-Pipelined (or non-blocking or symmetric) Hash Join (DPHJ) [Raschid and Su, 1986, Wilschut and Apers, 1993, Urhan and Franklin, 1999, Ives et al., 1999] that fully exploits pipeline and parallelism is widely adopted to reduce the response time. DPHJ maintains two hash tables for A and B respectively. When a tuple of one table, assumed A, arrives, this tuple is inserted into the hash table of A, and at the same time probed against the hash table of B to find the matching tuples. DPHJ is able to continue the join process unless the tuples of both tables are delayed. Also it can deliver the results of a query as soon as possible.

3.2 Distributed SPARQL Query Engines

The differences between the Web of Data and distributed DBMS bring specific challenges to distributed SPARQL query processing. Compared to a distributed DBMS, the Web of Data contains a much larger number of SPARQL endpoints whose ability are constrained to query answering, and collecting service descriptions on a large scale is not easy. Given that, some query engines, such as NetworkedGraph [Schenk and Staab, 2008], SemaPlorer [Schenk et al., 2009] and FedX [Schwarte et al., 2011] do not rely on service descriptions, while others, such as DARQ and DSP, maintain local files which contain basic statistics of SPARQL endpoints. Furthermore, sophisticated indexing techniques are used to describe LD. Stuckenschmidt et al. [2004] proposed an index structure based on property chain. Harth and Decker [2006], Neumann [2008] and Fletcher and Beck [2009] proposed different approaches based on B^+-tree that index all possible combinations of subject, predicate and object. Harth et al. [2010] used QTree to build RDF

data summaries. A good review of indexing techniques in SPARQL data management is provided by Staab [2011]. In addition, tools like RDFStats [Langegger and Woss, 2009] are available for collecting statistics of RDF data. The above approaches are able to provide detailed service descriptions, however, they cost much to build and cannot be widely reused between different query engines. To this end, approaches such as SPARQL 1.1 Service Description [Williams and Institute, 2011], and especially, the Vocabulary of Interlinked Datasets (VoID) [Alexander et al., 2009] are proposed to provide general terms and patterns for describing RDF datasets. VoID describes datasets from four aspects: general information (e.g. title); access metadata (e.g. accessible URIs); structural metadata (e.g. statistics) and linkages to other datasets. Compared to those preceding indexing techniques, VoID can be easily released alongside the datasets, and discovered via links in the data it describes or from known VoID repositories. The interoperability and convenience of VoID attracts more and more attention from both data publishers and consumers. For example, Böhm et al. [2011] introduced a MapReduce-based tool named voiDgen which can generate VoID descriptions for Web-scale data. Meanwhile, SPLENDID [Görlitz and Staab, 2011], WoDQA [Akar et al., 2012] and LHD [Wang et al., 2013] consumes VoID for source selection and query optimisation.

Source selection is significantly affected by service descriptions. When no service description is available, source selection can be done by explicitly binding URIs of data sources to triple patterns (e.g. NetworkedGraph and SemaPlorer), or by sending an *ASK* query to RDF datasets to check the existence of triple patterns in RDF datasets (e.g. FedX). The result of an ASK query is "true" if the triple patterns in the query exist in the dataset. The latter approach is accurate, but produces an equal amount of query requests as well as directly evaluate each triple pattern as a normal query against all data sources. Relevant sources also can be identified by analysing their metadata. A widely used approach is predicate matching, where data sources having the same predicate of the triple pattern are considered as relevant. Consequently, this method requires that the location of predicates is recorded. A reason for matching predicates rather than subjects or objects is that in general the number of distinct predicates is much smaller than that of distinct subjects or objects [Hu et al., 2011b][2]. DARQ, SemWIQ [Langegger et al., 2008], DSP and LHD employ predicate matching by examining their description

[2]Another (probably more relevant) evidence is given by the statistics of the Billion Triple Challenge 2012 (BTC2012) dataset which are available at `http://gromgull.net/blog/2012/07/some-basic-btc2012-stats/`.

files. If the vocabularies or ontologies of the data sources are known, analysing the classes to which subjects, predicates and objects belong also helps eliminate irrelevant sources. Furthermore, Akar et al. [2012] proposed twelve rules that enable sophisticated source selection by carefully examining VoID files. SPLENDID [Görlitz and Staab, 2011] adopts a mixed approach, that it firstly analyses the VoID files of data sources and then checks the existence of those triple patterns that are not bound to any data source. Thus SPLENDID accurately selects relevant data sources with lower network overhead than a pure asking approach adopted in FedX.

Due to the even higher communication cost on the Web than in distributed DBMS, many specific optimisation algorithms have been proposed to make distributed SPARQL query processing more economic and efficient. Most of these algorithms are closely related to (iterative) dynamic programming or greedy algorithms. For example, DARQ, SPLENDID and LHD adopt (iterative) dynamic programming in their query optimiser.

In the meantime, Stocker et al. [2008], Vandervalk et al. [2009] and Wang et al. [2011] (DSP) combine graph theory into query optimisation. They regard a Basic Graph Pattern (BGP) as a graph and use minimum-spanning-tree (MST) algorithms (which belong to greedy algorithms) to search for the optimal QEP[3]. Given a BGP subjects and objects are regarded as vertices and predicates are edges. The weight of each edge is set to the cost required to evaluate the edge given by the cost model. The optimal QEP of the BGP is produced in the process that generates the MST of the corresponding graph. The algorithm adopted by Vandervalk et al. [2009] and DSP firstly selects a concrete vertex (i.e. not a variable) and adds the minimum edge connected to that vertex to the MST[4]. Then the MST grows by adding the minimum edge that has only one vertex in the MST (i.e. one vertex of the edge must be outside of the MST). Triple patterns that correspond to edges in the MST are executed following the same order that those edges are added to the MST. During the execution there may be triple patterns whose both vertices have been bound to some values as a result of executing previous triple patterns. Such triple patterns correspond to edges whose their both vertices are in the MST but they are not included in the MST. They are used to filter intermediate results.

[3]Different from the other two, Stocker et al. [2008] take triple patterns, rather than subjects or objects of triple patterns, as the nodes of graphs. That is, the graph of Stocker et al. [2008] represents the relationship among triple patterns of a BGP while the graph of Vandervalk et al. [2009] and Wang et al. [2011] represent the BGP itself. Also, Stocker et al. [2008] didn't explicitly claim their algorithm, which adopted Prim's algorithm [Prim, 1957], as a MST algorithm.

[4]The algorithm can start with a variable vertex but doing so usually leads to more intermediate results.

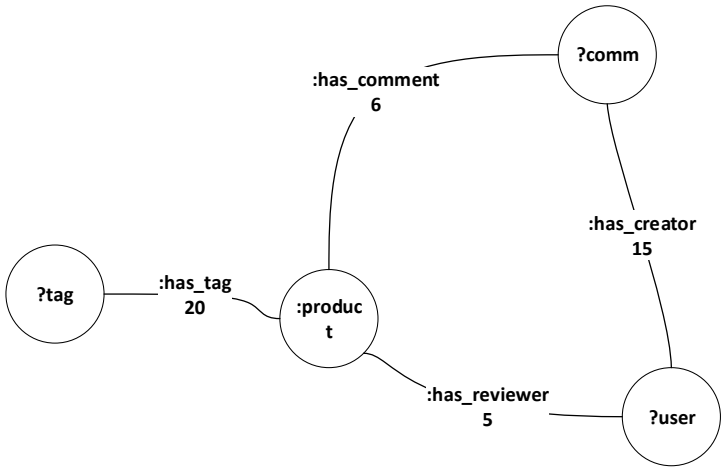


FIGURE 3.1: On each edge the top text gives the predicate and the bottom number gives the weight. The algorithm firstly selects the minimum triple pattern connecting to a concrete node and adds it to the MST. Then it selects the minimum triple pattern that connects to existing MST but also has a node outside the MST. Triple patterns with both nodes in the MST are used to filter intermediate results.

An example of the above process is shown in Figure 3.1. The algorithm consequently selects edge *:product :has_reviewer ?user* and *:product :has_comment ?comm*. Then the edge *?comm :has_creator ?user* becomes fully bound and is used to filter intermediate results. The edge *:product :has_tag ?tag* is selected in the end. No matter which MST algorithm is adopted, the key point is to select the minimum triple pattern at each step and triple patterns that are fully bound in previous execution are used to filter intermediate results.

Alongside the aforementioned algorithms, query rewriting heuristics, such as pushing down filters, is applied in optimisation as well. Several such rules have been described in Pérez and Arenas [2009].

The optimal plan is evaluated with respect to cost models, whose accuracy is effected by statistics of data sources. In distributed SPARQL optimisation, the selectivity-based cost model is used by most of SPARQL engines (OptARQ [Bernstein et al., 2007], Stocker et al. [2008], DARQ, DSP, SPLENDID). Usually the selectivity of a triple pattern is estimated as the product of the selectivity of subject, predicate and object of this triple pattern, based on statistics of RDF datasets, under the assumption that each part is independent and evenly distributed (unless extra knowledge is available about their distributions) [Bernstein et al., 2007, Stocker et al., 2008, Quilitz, 2008, Wang et al., 2011, Görlitz and Staab, 2011, Akar et al., 2012, Wang et al., 2013]. The required

statistics can be obtained form VoID files. If pre-computed statistics are not sufficient, heuristics are used to rank QEPs. For example, the variable-counting heuristics [Stocker et al., 2008] (also adopted in FedX) takes advantage of general experiences, however, its accuracy is arguable [Stocker et al., 2008].

The same as in distributed DBMS, execution of joins is critical for distributed SPARQL query execution as well, and many well-established join techniques are used in distributed SPARQL engines. Basic join implementations such as NLJ is used in DARQ for its simplicity of implementation, and HJ is used in DSP and SPLENDID to gain improved performance. The none-blocking join operator is used by Hartig and Bizer [2009] and variations of DPHJ (or SHJ) are used by Ladwig and Tran [2010, 2011], Acosta et al. [2011] to enable adaptive execution when data transfer on the Web is unstable. In order to reduce network traffic, bind join is adopted in DARQ, DSP, FedX and SPLENDID, while Semijoin is used by NetworkedGraphs [Schenk and Staab, 2008]. In DARQ and DSP, variables of triple patterns are replaced by intermediate results[5], and the bound triple patterns are evaluated instead of the original ones. This implementation is best used with engines that produce results in a streaming fashion (i.e. only one result is materialised at a time). Since only one result is used to bound a triple pattern at a time, it is simply extended with all results returned by executing the bound triple pattern to produce results for future execution. FedX (and SPLENDID adopts this method) transforms many bind joins as a long list of *UNION* clauses and therefore enables processing of many results in one request. NetworkedGraphs attach intermediate results as *FILTER* clauses with the original query to implement Semijoin. This approach can process many intermediate results with one query request, but the returned results have to be joined with those results used as filters. Under certain conditions, bind join and Semijoin can reduce communication cost, however, result in longer execution time since the operators being joined are executed sequentially.

As a summary, the choices of optimisation techniques of the most popular engines are listed in table 3.1.

Apart from the aforementioned distributed SPARQL query engines, document-oriented, keyword-based search engines are also present on the Web of Data, such as the ones by Watson [D'Aquin et al., 2007], Sindice [Oren et al., 2008], Falcons [Cheng and Qu, 2009]

[5]Here a result refers to a *solution mapping*, that maps a variable to a value [Prud'Hommeaux and Seaborne, 2008].

Table 3.1: Optimisation techniques of popular query engines

	DARQ	**DSP**	**FedX**	**SPLENDID**	**LHD**
Service descriptions	Local files	Local files	None	VoID	VoID
Source selection	Predicate matching	Predicate matching	ASK query	Predicate matching, Asking	Predicate matching, ASK query
Cost model	Selectivity based	Selectivity based	Variable counting heuristics	Selectivity based	Selectivity based
Opt. algorithm	IDP	MST	Heuristics	IDP	IDP, Heuristics
Join	NLJ, BJ	NLJ, BJ	HJ, BJ	HJ, BJ	HJ, BJ

and SWSE [Hogan et al., 2012]. These semantic web search engines work in the same way as typical web search engines (e.g. Google) by crawling RDF data and building indices for quick looking up. Semantic web search engines could be a convenient way to locate a piece of RDF data. However, since their support for SPARQL is limited or even not available, they are not the right tool to answer distributed SPARQL queries at the moment of writing.

3.3 RDF Store Benchmarking

The rapid growth of LD not only offers potentials for efficient distributed SPARQL engines, but also raises demands for benchmarks that compare performance of query over LD. Consequently, benchmarks covering different aspects of RDF stores have been proposed. Here we roughly divide popular benchmarks into two categories based on their assessment objectives:

- For comparing performance of reasoning: Lehigh University Benchmark (LUBM) [Guo et al., 2005], University Ontology Benchmark (UOBM) [Ma et al., 2006] which extends LUMB, and JustBench [Bail et al., 2010].

- For comparing performance of query processing: SP^{2}Bench [Schmidt et al., 2009], DBpedia SPARQL Benchmark [Morsey et al., 2011], Berlin SPARQL Benchmark (BSBM) [Bizer and Schultz, 2009], FedBench [Schmidt et al., 2011], and Social Network Intelligence BenchMark (SIB) [Boncz et al.].

Since we examine approaches that focus on improving query efficiency, benchmarks aiming to compare reasoning performance are out of scope. Therefore, we only review benchmarks of the second category in this section.

SP^{2}Bench focuses on testing typical features and operators of the SPARQL language. Its dataset is based on the syntax of the DBLP database[6], and a data generator is provided to generate arbitrarily large data. The query mix of SP^{2}Bench covers the typical structure of SPARQL queries (e.g. star-shaped or chain-shaped queries).

Meanwhile, BSBM, which supersedes the DBpedia SPARQL Benchmark, is based in a business use case in which customers review products having various features and from different vendors. Its dataset contains the following classes: *Product*, *ProductType*, *ProductFeature*, *Producer*, *Vendor*, *Offer*, *Review*, and *Person*. An overview of the data model is shown in figure 3.2. BSBM also provides a scalable data generator. The queries of BSBM reflex real-world requirement and mix different features and patterns of SPARQL. Beside the benchmark itself, the authors (Christian Bizer and Andreas Schultz) consequently published benchmark results for most existing RDF stores via various sources (e.g. blog, web page, mailing list). This potentially encouraged many other researchers to publish BSBM results as well, and makes BSBM one of the most widely used benchmarks[7].

Existing research suggests that aforementioned benchmarks (namely LUMB, SP^{2}Bench and BSBM) are relational-like and do not represent structure of real RDF datasets [Duan and Kementsietsidis, 2011]. To this end, benchmarks simulate or use real world datasets have been proposed as well. Especially, social network data attracts much attention due to their graph structure. SIB simulates a social network scenario using a data generator called S3G2 [Pham et al., 2013]. Meanwhile, Przyjaciel-Zablocki et al. [2013] argues that SIB is still short of being realistic, and proposes a SPARQL 1.1 benchmark using

[6] `http://www.informatik.uni-trier.de/~ley/db/`

[7] A list of benchmarking results is available at the RDF Store Benchmarking page `http://www.w3.org/wiki/RdfStoreBenchmarking`, in which BSBM results take a large proportion and most up to date positions.

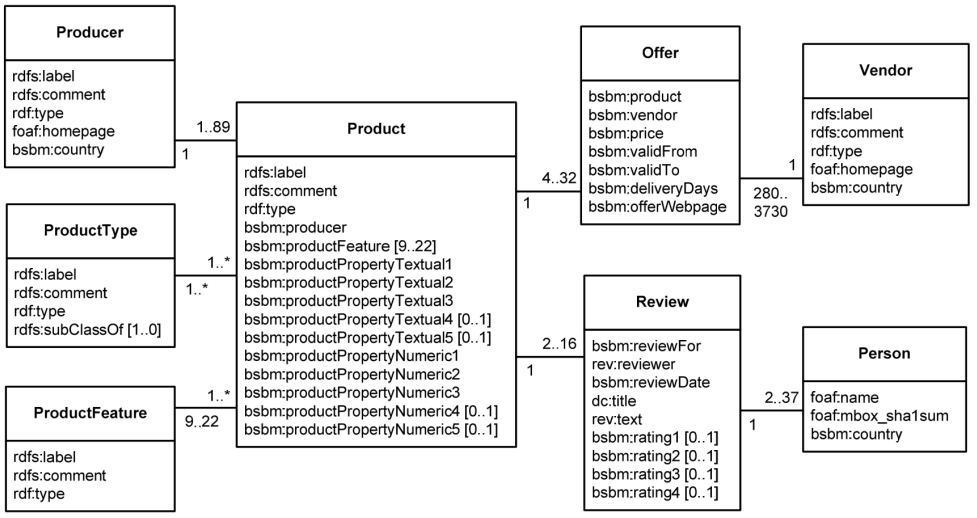


FIGURE 3.2: Overview of the BSBM data model [Bizer and Schultz, 2009]

real world social network data. Moreover, FedBench, which is designed to evaluate distributed queries, adopts real world data from multiple domains.

People may argue that real world data presumably has advantages of validity over artificial data. However, carefully designed artificial data, such as those provided by BSBM, can reflex the real world, and their validity is approved by the wide acceptance of the SPARQL community. Besides, artificial data are more flexible in terms of scalability and extension capability, and help set up various environments more easily than real world data. Especially for distributed query benchmarking, once real-world-based datasets are chosen, they determines the contents as well as the scale and data distribution of the RDF network the benchmark simulates. In case of evaluating features that are not covered by the benchmark's original design (e.g evaluating queries with co-reference), artificial data can be extended with less efforts than real world data. To this end, we base the evaluations of this thesis in an evaluation framework that uses artificial datasets. Details of the framework are given in chapter 4.

Chapter 4

DSEF: A Distributed SPARQL Evaluation Framework

In this thesis we investigate techniques that are promising to improve efficiency of distributed queries in environment with or without co-reference. To compare the efficiency of our approaches with existing ones, we developed a distributed SPARQL evaluation framework (DSEF) that is capable of evaluating query engines in distributed settings and with co-reference. Due to the unique requirement of co-reference, DSEF is based on well-established artificial data (BSBM data), by which we keep in line with existing benchmarking approaches.

4.1 Overview of the Evaluation Framework

DSEF is tailored for evaluating distributed SPARQL engines in networks of arbitrary scales, and with co-reference taken into account.

DSEF provides a virtual-machine-based network architecture that can conveniently simulate networks of different sizes and computing power. Together with artificial data, this architecture enables creation of LD networks having arbitrary numbers of endpoints and triples. In addition, the distribution of data among endpoints is easily controlled using a set of tools contained in DSEF.

For interoperability reasons we adopt the well-established dataset of BSBM. Besides, we statistically investigate co-reference in the real world, based on which *owl:sameAs* statements are generated. The assessment queries used in the framework are as well based on BSBM queries. The original BSBM queries are designed to evaluate various aspects of RDF stores and some features can introduce undesired disturbance to the performance of distributed SPARQL engines. In DSEF, the BSBM queries are modified in a way that retains the semantic of the original queries but prevents undesired performance disturbance. The modification is neutral to all the engines that will be evaluated in this thesis. Details of the assessment queries are given in section 4.3.

The framework tool set is able to 1) generate structured RDF data of an arbitrary size; 2) generate co-reference statements based on real-world proportion and distribution; 3) divide a large amount of data into smaller pieces w.r.t specific distributions; 4) generate detailed VoID descriptions; 5) efficiently dispatch data to remote RDF stores; 6) automatically evaluate distributed SPARQL engines and collect desired testing statistics.

The architecture of DSEF is shown in figure 4.1. In the following we provide details of datasets, assessment queries and each framework tool. In addition, we describe an environment based on DSEF, in which evaluations in this thesis are performed.

4.2 Assessment Data and Co-Reference Generation

DSEF adopts the dataset of BSBM. Furthermore, to evaluate the performance of distributed SPARQL engines with the presence of co-reference, DSEF extends BSBM data with *owl:sameAs* statements w.r.t real-world statistics. Existing research implies that co-reference follows a power law distribution [Ding et al., 2010, Hu et al., 2011b], but no explicit evidence is given. We analyse the data of Billion Triple Challenge (BTC) 2012[1] for statistics of co-reference. The BTC data is crawled from all LD, and can be regarded as a snapshot of the entire LD cloud.

There are in total 1.4 billion triples of the BTC 2012 dataset[2]. 0.00246% of them, which is equal to 3449341 triples, are *owl:sameAs* statements. We divided resources into categories w.r.t to the number of co-reference relationships they have. That is,

[1] `http://challenge.semanticweb.org/`

[2] This is confirmed by the result given by `http://gromgull.net/blog/category/semantic-web/billion-triple-challenge/`

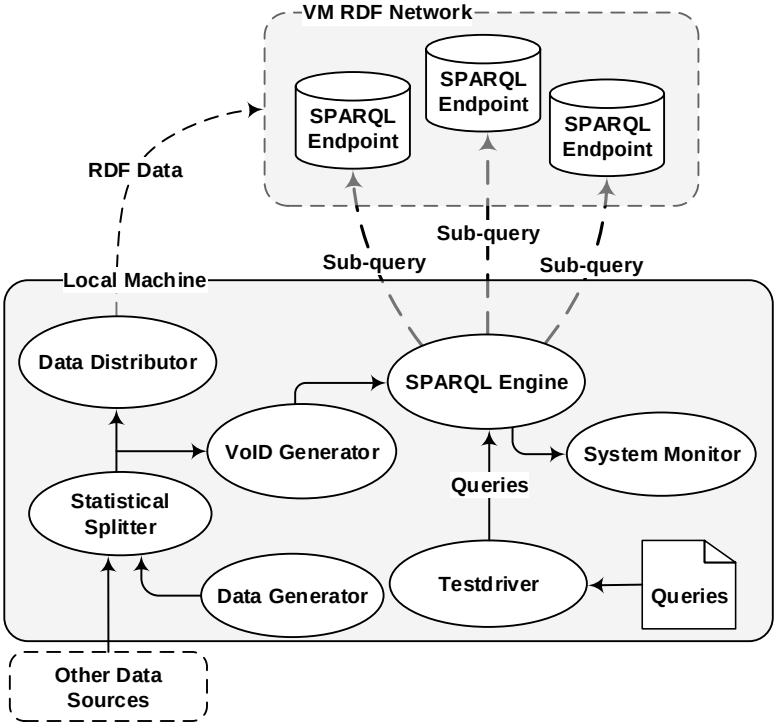


FIGURE 4.1: Solid arrows denote local data flow while dotted arrows denote remote data flow. The *data splitter* splits data from either the *data generator* or the LD cloud according to a given distribution. Then the *data distributor* uploads data to SPARQL endpoints. After the endpoints are ready, the *testdriver* reads queries from a file and calls the distributed SPARQL engine to process these queries. The engine processes the queries and returns results to the *testdriver*. The *testdriver* records the time of query processing and generates a performance report. Meanwhile the *system monitor* records the memory and CPU usage and the network flow.

each category contains resources that have occurred in a certain number of *owl:sameAs* statements. We accumulate the number of resources of each category, and produce the diagram shown in figure 4.2. We find that points in figure 4.2 are approximate to a power law distribution $p(x) = \alpha x^{-\beta}$, where $\beta = 2.528$. The aforementioned percentage and the distribution function are used by DSEF to generate co-reference for given datasets. Later in the evaluation of this thesis a dataset of 70 million triples is used, and 0.18 million (0.0026) *owl:sameAs* statements are generated accordingly.

Generation of co-reference is achieved by linking resources using *owl:sameAs*. To reproduce the distribution of real-world co-reference, we use a power law random number generator. It accepts two parameters which are the power law exponent $\beta = 2.528$ and the number of elements (i.e. distinct resources that have co-reference). For a given resource, we use this generator to decide the number of *owl:sameAs* statements that

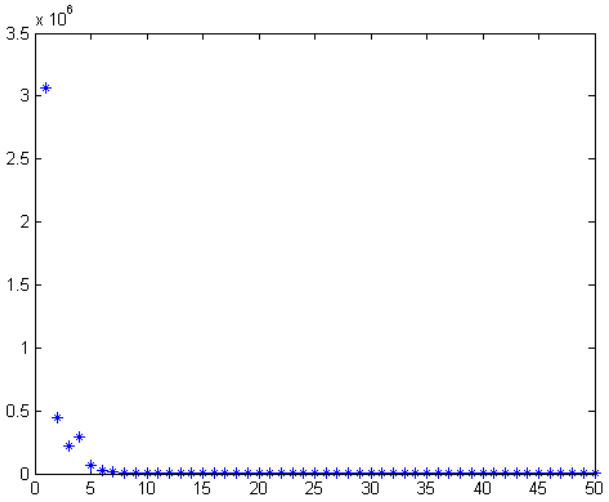


FIGURE 4.2: The horizontal axis presents categories of URIs (as subjects or objects) having 5, 10, 15 ... co-reference respectively, while the vertical axis presents the number of resources falling in each category.

link this resource with other randomly chosen resources. We also take into account that resources of BSBM data fall into different classes as shown in figure 3.2. We generate co-reference for each class separately to make sure that resources are only equivalent to those in the same class. Furthermore, numbers of co-reference that are larger than the total number of instances of a class (very rare) are discarded, and new ones are picked.

It should be noticed that our method focuses on reproducing the deviation of VoID statistics, and the number of equivalent URIs of resources, those aspects that are closely related to query optimisation. Like other artificial data, co-reference statements generated by our method are different from real-world data, but simulate co-reference in the real linked data cloud in order to test the efficiency of optimisation techniques developed in the real environment.

4.3 Assessment Query Set

The query mix of BSBM is designed to emulate real world use cases, and consequently contains complex queries having multiple BGPs. In the meantime, most distributed SPARQL engines (including all engines that will be evaluated in this thesis) perform optimisation on BGPs rather than whole queries. Results of BGPs are aggregated by internal functionalities of platforms on which the engines are built. For queries having

more than one BGPs, incompatibility may exist between platforms that use single-thread aggregation (e.g. Jena[3]) and engines that use parallel optimisation and execution (e.g. LHD). For instance, within a BGP LHD intensively uses parallelisation to produce many results simultaneously, however, only one result is passed from one BGP to another at a time by Jena. To this end, we transform BSBM queries into queries that are compatible with parallel engines, in a way that the semantics of the original queries are preserved. The following rules are used in this procedure:

- Queries initially having one BGP are left untouched (e.g. BSBM query 1).
- The *UNION* keyword is compatible with parallel engines in both Sesame and Jena. Therefore queries having only *UNION* keywords are left untouched (e.g. BSBM query 4 and 11).
- The *OPTIONAL* keyword is incompatible with parallelisation. If the BGP following a *OPTIONAL* keyword has matching results, it is merged into the main BGP (the one enclosing the optional BGP). Otherwise, the optional BGP is removed (e.g. BSBM query 2, 7, 8).

By applying the aforementioned rules, optional results, if there are any, are merged into mandatory results. If there is no optional result, query results remain unchanged.

In addition, all *FILTER* expressions are removed as well. This is because: 1) *FILTER* expressions break a BGP into small pieces, and affect query performance in an undetermined fashion; 2) there is mature research [Pérez and Arenas, 2009] on optimisation of *FILTER* expressions (i.e. rewriting queries using *FILTER* values), which can be applied on top of any other optimisation techniques. Removing *FILTER* helps solely revealing the performance of other techniques; and 3) none of the engines that will be evaluated in this thesis claims adoption of *FILTER* optimisation. Removing *FILTER* expressions will not introduce inequity.

The assessment queries of DSEF are achieved after applying all aforementioned modifications. These queries especially emphasise the BGP optimisation and execution efficiency of distributed SPARQL engines. A complete list of these queries is given in appendix A.1.

[3]In detail, Jena streams results of one BGP to another, that only one result is passed at a time. Consequently the optimisation and execution of the second BGP are constrained to single-thread.

4.4 Assessment Metrics

DSEF focuses on assessing the efficiency of distributed SPARQL engines, while also aiming to provide insight of optimisation and execution techniques adopted by query engines. In most SPARQL benchmarks (e.g. SP^{2}Bench, BSBM, FedBench) the number of queries executed in a certain time period is used to measure the efficiency of query processing. Query efficiency is jointly determined by the engines' abilities of reducing the size of network traffic and increasing the average data transmission rate. Therefore, we further include network traffic, and transmission rate in the metrics of DSEF. The following metrics are regarded as primary metrics in DSEF:

- Query per second (QPS), represents the average number of queries executed per second.
- Network traffic, represents the total amount of network traffic (both incoming and outgoing) produced due to executing queries.
- Transmission rate, represents the average speed of network communication. It is calculated as the network traffic divided by the query execution time.

In the meantime DSEF also provides two secondary metrics, which are not included in existing SPARQL benchmarks:

- CPU usage, presents the average percentage of CPU used to execute a certain query.
- Memory usages, presents the average amount of memory used to execute a certain query.

The secondary metrics are not used to compare the effectiveness of techniques adopted in query engines. This is because CPU and memory usage are not optimisation objectives of the engines under testing. Rather, the secondary metrics are used to verify that the amount of system resources required by tested engines are practical.

Beside performance, DSEF also takes care of the correctness of query execution of engines under testing. We prepare a RDF store that have all assessment data in it, and execute

the assessment queries against this centralised store. By that we obtain the correct result of each query. Before evaluating an query engine we perform test runs and make sure all engines give the correct answer. Thus we prevent engines to quickly return results that are not correct.

4.5 The Framework Tool Set

The evaluation framework contains five tools: *data generator*, *statistical splitter*, *VoID generator*, *data dispatcher* and *test driver*. In general, all tools are designed to be adaptive and scalable (when required). Using these tools, RDF networks containing large data can be set up conveniently. Details of each one is given below.

Data generator

The *data generator* extends the BSBM data generator with functionality of co-reference generation. It first calls the BSBM generator to create RDF data of a certain size, then uses the methods described in section 4.2 to generate corresponding instances of co-reference.

As stated in section 3.3, the BSBM generator is built on a carefully designed data model that reflects a business use case in which customers review products having various features and from different vendors. The BSBM data has received a wide acceptance in the LD community[4]. Co-reference is generated by following real world statistics and distribution, as described in section 4.2. Both parts are reflective of the actual structure of real world LD.

Generating co-reference within a certain class (e.g. *Product*) requires that all instances of the class are available before creating *owl:sameAs* statements. For large datasets it is impossible to hold instances of all classes in memory (sometimes even a single class does not fit in memory). In order to gain efficiency and scalability at the same time, the *data generator* firstly scans the given data only once to extract instances of all classes and stores them on hard disk. This procedure employs streaming techniques and requires

[4] A list of benchmarking results is available at the RDF Store Benchmarking page `http://www.w3.org/wiki/RdfStoreBenchmarking`, in which BSBM results take a large proportion and most up to date positions.

only a (small) constant amount of memory. Then the generator reads back instances of one class at a time to generate co-reference. The hard disk is used as secondary storage if a class contains more instances than the main memory can hold.

Statistical splitter

The *statistical splitter* accepts a RDF data file and a distribution function. It splits the data into certain numbers of smaller pieces w.r.t to the given distribution. The supported distributions include uniform distribution, normal distribution and power-law distribution which is common on the Web.

The splitter scans through the given data file and determines the destination (i.e. a data piece) of each triple based on the number given by an internal random number generator. The frequency of the number of each data piece is determined by the given distribution. This procedure requires constant memory since no data needs to be stored.

Data dispatcher

The *data dispatcher* is used to dispatch data to remote datasets. It accepts a file containing URIs of RDF datasets, and a corresponding list of RDF files. For efficiency multiple threads are used to upload RDF data to remote datasets. Furthermore, the dispatcher keeps records of the number of triples that have been uploaded. In case of connection issues, interrupted uploading can be resumed according to those records. This feature is necessary to upload a large number of triples.

VoID generator

The *VoID generator* produces VoID descriptions for given RDF data. The generated VoID description contains the URI of the SPARQL endpoint of given data, which indicates a potential query target. Furthermore, it contains the following statistics of each dataset: 1) the total number of triples, distinct resources and distinct predicates; 2) for each predicate, the number of triples, distinct subjects and objects. The latter statistics are presented as property partitions. An example VoID description is shown in figure 5.2.

Test driver

The *test driver* accepts a file of testing queries and a distributed SPARQL engine. It automatically reads the queries and executes them using the given engine. Since the responding time of SPARQL endpoints can get slower for continuous query execution, breaks are left between the execution of each query (so that the endpoints can recover). Currently we set a 10 second interval between runs of the same query, and a 10 minute interval between execution of different queries based on previous experiences. For each query the *test driver* records the average execution time and calculates the QPS. It also records the average size of results. The result size is used to confirm that all engines under testing return the same results, and to calculate the extra results led by co-reference. The output of the *test driver* is a "csv" file that can be further processed. In the meantime, the total network traffic of each query is recoded. With the responding time we calculate the average transmission rate of a certain engine on a certain query. Finally, we also record the CPU and memory usage of tested engines.

When distributed SPARQL engines are assessed using the framework, firstly RDF data are obtained from the LD cloud or generated by the *data generator*. Once the data are ready, it is split into pieces by the *data splitter* according to a certain distribution, and also basic statistics of each piece of data can be collected by the *statistics collector*. After that, the *data distributor* dispatches each piece of data to certain remote datasets. Then the distributed SPARQL approach can be evaluated by the *test driver* against those remote datasets.

DSEF extends the widely accepted benchmark BSBM for evaluating distributed SPARQL engines with the presence of co-reference, and is able to simulate RDF networks of arbitrary sizes. An initial version of DSEF has been published in Wang et al. [2011] and used for evaluating DSP and LHD.

Chapter 5

Querying LD with Detailed VoID Statistics

Given a distributed SPARQL engine, its efficiency results from the compounding of techniques adopted in all four stages of distributed SPARQL query processing. The effectiveness of query optimisation techniques usually depends on the environment in which they are applied. In other words, if a technique works well in certain environments, the same effectiveness is not guaranteed under other circumstances[1]. Therefore, it is necessary to distinguish different environments before determining the techniques we use for executing distributed SPARQL queries efficiently.

As stated in chapter 2, source selection and query optimisation are closely related to service descriptions. To this end, we distinguish two typical circumstances of the LD cloud w.r.t different accuracy of service descriptions. We assume that, on a large scale, it is preferred to obtain service descriptions from VoID files provided by each SPARQL endpoint, than to maintain private indices and statistics for all data sources. Given that assumption, we examine two typical scenarios in which VoID with different accuracy are available. For each scenario we propose a scheme of techniques that aims to improve the efficiency of distributed SPARQL query processing, and develop a distributed SPARQL

[1]This is a weaker implication of the *No Free Lunch (NFL)* theorem of search and optimisation, which states that any two optimization algorithms are indistinguishable over all possible problems [Wolpert and Macready, 1995, 1997]. Intuitively, no algorithm is good at solving all problems.

engine based on the scheme. In this chapter we present the scheme and the corresponding engine called LHD-s that work with detailed statistics. The other scheme will be described in chapter 7.

5.1 Overview of LHD-s

As detailed in chapter 3, VoID can give statistics of either a whole dataset or its partitions (either class partitions or property partitions). In most SPARQL queries, predicates are explicitly given (except those having only variable predicates), while classes (of resources in the queries) are not always known[2]. Consequently, LHD-s is designed to work primarily with statistics of property partitions while it can also benefit from statistics of class partitions.

It is worth mentioning that at this moment VoID documents with statistics are only provided by a few datasets[3]. However, publishing detailed statistics in VoID would be straightforward once there are demands. In the case that detailed VoID statistics are not available, we will show in chapter 7 and 8 that efficient query processing can be achieved by exploiting runtime statistics.

With detailed VoID descriptions, it is possible to make reasonable cost estimation for queries, and thus it is worth using optimisation algorithms that produce high quality QEPs. For best results, LHD-s adopts a dynamic-programming-based approach in query optimisation. Furthermore, LHD-s follows the static optimisation approach since 1) the cost of any QEP can be estimated before query execution; 2) dynamic programming generates complete QEPs before query execution. In order to increase efficiency of query execution, we propose an execution system that employs parallelism at different levels. With this execution system LHD-s is able to maximumly exploit the bandwidth of data sources without overloading them. The architecture of LHD-s is shown in figure 5.1. Details of each component of LHD-s are given in the following sections.

[2]The classes of resources of a query can be known if there are triple patterns having *rdf:type* predicates and concrete objects. Or, if schema of predicates are given, classes can be inferred from the domain and range of predicates. However, neither is as common as the way in which a predicate is given in queries.

[3]Example VoID documents can be found at `http://void.rkbexplorer.com/`.

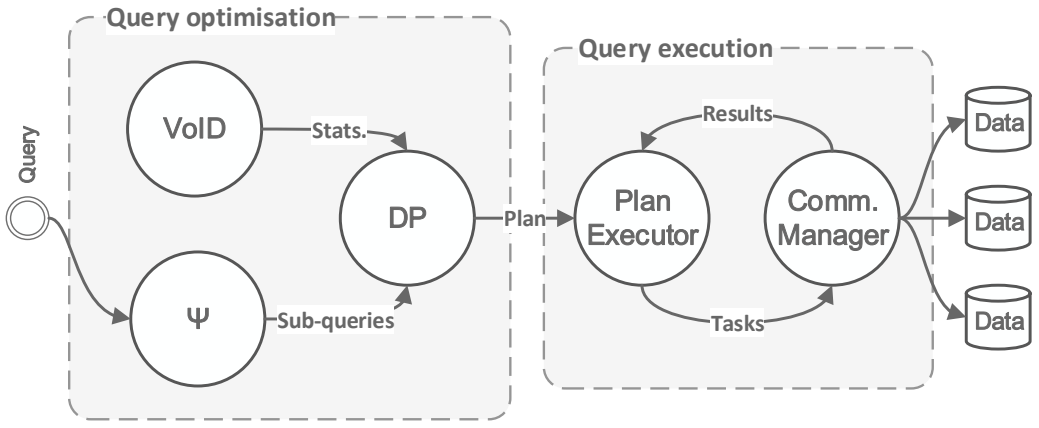


FIGURE 5.1: Given a SPARQL query Ψ first breaks it into independent sub-queries. Each of these sub-queries is optimised by a dynamic programming (DP) based algorithm using VoID statistics. The optimal QEP is executed by a plan executor, which does not contact datasets directly, but submits query tasks to a communication manager during the execution. The communication manager maintains physical connection to each dataset individually.

5.2 VoID Service Descriptions

To gain best results of LHD-s query optimisation, we generate VoID files that contain as detailed statistics as VoID can provide for all data sources. These statistics contain the numbers of distinct subjects and objects, as well as the total number of triples, per predicate partition. An example is given in figure 5.2. Although such VoID files are not available from all SPARQL endpoints in the LD cloud, they are used here as the upper bound of statistics that are nowadays possible to get from SPARQL endpoints. Similar statistics are also required by SPLENDID Görlitz and Staab [2011].

The example VoID file describes a dataset whose URI is d (line 1). This dataset contains t_d triples (line 4), s_d distinct subjects (line 5) and o_d distinct objects (line 6). In addition, there is a predicate p in d that is described by a property partition (line 8-13). In this property partition, the number $t_{d.p}$ (line 10) indicates how many triples in d are having p as predicate. In addition, $s_{d.p}$ and $o_{d.p}$ respectively give the numbers of distinct subjects or objects associated with p. The following relations hold for these statistics:

$$t_d = \sum_{p \in d} t_{d.p},\ s_d \leq \sum_{p \in d} s_{d.p},\ o_d \leq \sum_{p \in d} o_{d.p} \tag{5.1}$$

```
1: d a void:Dataset ;
2:       ...
3: # simple statistics:
4:       void:triples "t_d" ;
5:       void:distinctSubjects "s_d" ;
6:       void:distinctObjects "o_d" ;
7: # statistics per predicate:
8:       void:propertyPartition [
9:            void:property p ;
10:           void:triples "t_d.p" ;
11:           void:distinctSubjects "s_d.p" ;
12:           void:distinctObjects "o_d.p" ;
13:      ], [
14:      ...
15:      ].
```

FIGURE 5.2: Statistics in a VoID file

For simplicity, here $p \in d$ means p is a predicate in d. The first equation holds because each triple belongs and only belongs to a predicate partition. For the later two, a subject or an object may occur in multiple predicate partitions, and the number of distinct subject/object of the dataset is no more than the sum of the numbers of distinct subject/object of all predicate partitions.

VoID can also contains class partitions and corresponding statistics. However, class partition statistics are mostly used to optimise only triple patterns with the *rdf:type* property and bound objects. In this thesis we do not use class partitions since they are not as widely used as property partitions.

5.3 Data Source Selection

Data source selection eliminates irrelevant data sources at an early stage and thus increases the accuracy of cost estimation. LHD-s adopts a two-phase source selection that is used in SPLENDID. First, LHD-s analyses predicate partitions in VoID files. Data sources having the same predicate of a triple pattern are identified as relevant candidates. Second, *ASK* queries enclosing the triple pattern are sent to these candidates, and data sources that give positive response are kept.

5.4 Cost Estimation

Most existing distributed SPARQL engines (e.g. DARQ, FedX and SPLENDID) use cost models that estimate the number of (intermediate) bindings generated during query execution. Here we propose a different cost model to estimate the responding time of a QEP, to cope with LHD-s' parallel execution system. Given a QEP, the basic operation is the execution of triple patterns. Executing a triple pattern involves sending one or more query requests, with or without pre-computed bindings, to all relevant endpoints, and receiving corresponding results. We assume that the time of sending requests to or receiving responses from SPARQL endpoints, is proportional to the number of bindings enclosed in the communications. Using statistics of VoID files, we first estimate the cardinality of outgoing and incoming bindings, and then estimate the response time of a QEP.

From the VoID file shown in figure 5.2, we can have the total number of triples t_d, distinct subjects s_d and objects o_d in d. We can also have the number of triples $t_{d.p}$, distinct subjects $s_{d.p}$ and objects $o_{d.p}$ in the partition of p. We assume that subjects and objects are uniformly and independently distributed in data sources. In the following a question marked letter (e.g. ?x) denotes a variable, a lower-case letter (e.g. s) denotes a concrete value, and an upper-case letter (e.g. O) denotes either a variable or a concrete values. Given a triple pattern $T : \{S\ P\ O\}$, we define a function $src(T)$ that gives the set of relevant data sources of T. We use $sel(T, x)$ and $card(T, x)$ to denote the selectivity and cardinality of $x \in \{S, P, O\}$ w.r.t T respectively. It is worth noticing that the same x can have different selectivity and cardinality in different triple patterns.

5.4.1 Cardinality of a Single Triple Pattern

Given a single triple pattern $T = \{S\ P\ O\}$, the selectivity of each part is estimated following the approach used in [Stocker et al., 2008, Quilitz, 2008], as follows:

$$sel(T,S) = \begin{cases} \frac{1}{\sum_{d \in src(T)} s_d} & \text{if } var(P) \wedge \neg var(S), \\ \frac{1}{\sum_{d \in src(T)} s_{d.p}} & \text{if } P = p \wedge \neg var(S), \\ 1 & \text{if } var(S). \end{cases} \tag{5.2}$$

$$sel(T,P) = \begin{cases} \frac{\sum_{d \in src(T)} t_{d.p}}{\sum_{d \in src(T)} t_d} & \text{if } P = p, \\ 1 & \text{if } var(P). \end{cases} \tag{5.3}$$

$$sel(T,O) = \begin{cases} \frac{1}{\sum_{d \in src(T)} o_d} & \text{if } var(P) \wedge \neg var(O), \\ \frac{1}{\sum_{d \in src(T)} o_{d.p}} & \text{if } P = p \wedge \neg var(O), \\ 1 & \text{if } var(O). \end{cases} \tag{5.4}$$

where $var(X)$ is a function that returns $true$ if X is a variable or $false$ otherwise. Assuming that $sel(T,S)$, $sel(T,P)$, and $sel(T,O)$ are statistically independent [Selinger et al., 1979, Christodoulakis, 1984], the selectivity of the triple pattern T is estimated as $sel(T) = sel(T,S) \cdot sel(T,P) \cdot sel(T,O)$. The cardinality of T is estimated as

$$card(T) = \sum_{d \in src(T)} t_d \cdot sel(T) \tag{5.5}$$

For a triple pattern having a variable subject and object, the estimated cardinality is accurate (equals to $\sum_{d \in src(T)} t_{d.p}$). Since we consider only the relevant data sources of T (rather than the "global graph" constructed as the union of all data sources), better source selection can increase the accuracy of the estimation.

5.4.2 Cardinality of Joined Triple Patterns

Estimating the cardinality of joined two triple patterns can be difficult without join selectivity, defined as:

$$sel(T_1 \bowtie T_2) = \frac{card(T_1 \bowtie T_2)}{card(T_1) \cdot card(T_2)}$$

Join selectivity are not available from VoID files, and can be costly to maintain. Two triple patterns can join on subject-subject (SS), subject-object (SO), object-subject (OS) and object-object (OO). Since the join order is insignificant here, for n distinct predicates, a total $2n^2 + n$ records of join selectivity need to be stored[4].

To walk around join selectivity DARQ and SPLENDID group triple patterns with the same subject together and apply the following method. Given three triple patterns $T_1 : \{?x\ p_1\ o_1\}$, $T_2 : \{?x\ p_2\ o_2\}$ and $T_3 : \{?x\ p_3\ ?y\}$, $card(T_1 \bowtie T_2)$ is estimated as $min(card(T_1), card(T_2))$, and $card(T_1 \bowtie T_3)$ is estimated as $card(T_1) \cdot card(T_3)$. For the first estimation to hold it requires that the domain of p_1 is a subset or a superset of the domain of p_2[5]. The second estimation requires that the domain of p_1 is a subset of the domain of p_3. However, without schema of properties (e.g. domains and ranges of properties) it is difficult to decide whether two properties commit to the above requirements.

In LHD-s we take advantage of the fact that the cardinality of joined two triple patterns is irrelevant to the join method. To approximate join selectivity we assume that two triple patterns $T_1 : \{?s\ p_1\ O_1\}$ and $T_2 : \{?s\ p_2\ O_2\}$ are joined using bind join. If T_1 is executed first, $card(T_1)$ intermediate results are produced and are used to execute T_2. Each intermediate provides a value of $?s$ (not necessarily distinct), and thus the result size of T_2 is estimated as $card(T_1) \cdot sel(T_2, s) \cdot card(T_2)$. This is also regarded as the cardinality of the join $card(T_1 \bowtie T_2)$, and the corresponding join selectivity is $sel(T_1 \bowtie T_2) = sel(T_2, s)$. Since the execution can be performed in the reverse order, the join selectivity can be $sel(T_1, s)$ as well. In order to combine both cases, we use the

[4]Stocker et al. [2008] have suggested that the number of records is $4n^2$. It could be that the join order is considered significant.

[5]A more precise requirement here is that the results of T_1 is a subset or a superset of the results of T_2. However, we cannot know that in advance from VoID files.

geometric mean $(sel(T_1, s) \cdot sel(T_2, s))^{\frac{1}{2}}$ as the join selectivity. For two arbitrary triple patterns $T_1 : \{S_1\ P_1\ O_1\}$ and $T_2 : \{S_2\ P_2\ O_2\}$ we have

$$sel(T_1 \bowtie T_2) = \begin{cases} \sqrt{sel(T_1, v) \cdot sel(T_2, v)} & \text{if } v \neq \phi, \\ 1 & \text{if } v = \phi. \end{cases} \tag{5.6}$$

where v is the join variable (but is regarded as concrete in estimation).

The cardinality of n joined triple patterns is estimated as a sequence of two-triple-pattern join

$$card(T_1 \bowtie T_2 \bowtie \cdots \bowtie T_n) = \prod_{i=1}^{n-1} sel(T_i \bowtie T_{i+1}) \cdot \prod_{i=1}^{n} card(T_i) \tag{5.7}$$

5.4.3 A Response Time Cost Model

As a result of intensive use of parallelism, LHD-s adopts a response time cost model rather than a network traffic model. To estimate a QEP we distinguish the execution of join that require pre-computed bindings (e.g. bind join, Semijoin, denoted as $(q \bowtie_B t)$, where q is a join or a triple pattern and t is a triple pattern) from those which do not need pre-computed bindings (e.g. hash join, nested loop join, denoted as $(q \bowtie p)$, where q and p are joins or triple patterns). Two triple patterns involved in a hash join can be executed in parallel while in a bind join they have to be executed in sequence. We call an access plan an *independent access plan* if it executes a triple pattern directly, or a *dependent access plan* if pre-computed bindings are used to execute a triple pattern[6]. We denote an independent access plan of t as $acc(t)$, and a dependent access plan with bindings of a sub-query q as $acc(q, t)$. We assume the response time of a query is proportional to the number of bindings sent to and returned from a data source, and the response time of a QEP is estimated using the following equations:

[6]It should be noticed that the execution of a dependent access plan also produces the results of a bind join.

$$cost(q \bowtie p) = max(cost(q), cost(p)) \tag{5.8}$$

$$cost(q \bowtie_B t) = cost(q) + cost(acc(card(q), t)) \tag{5.9}$$

$$cost(acc(t)) = rt_q + card(t) \cdot rt_t \tag{5.10}$$

$$cost(acc(q, t) = card(q) \cdot rt_q + card(q \bowtie t) \cdot rt_t \tag{5.11}$$

where rt_q is the time of sending a triple pattern or a pre-computed result to a data source, and rt_t is the time of receiving a result.

5.5 Identifying Independent Sub-Queries

LHD-s exploits parallelism intensively. We propose an algorithm that identifies independent sub-queries, each of which can be optimised and executed in parallel. This algorithm, named Ψ[7], increases the degree of parallelism in a way that network traffic is not increased. In addition, LHD-s adopts dynamic programming to produce optimal QEPs. However, dynamic programming can take a significant amount of time to optimise queries with many triple patterns. Since Ψ breaks a large query into smaller ones, the complexity of query optimisation is reduced without compromising the quality of QEPs. By combining the algorithm presented below and dynamic programming, LHD-s' optimiser can produce good quality QEPs within acceptable time.

SPARQL queries are composed by Basic Graph Patterns (BGPs), which are a set of conjunctive triple patterns. A BGP can be regarded as a connected graph that subjects and objects are nodes (or vertices) and triple patterns are edges. We observed that given two edges (triple patterns) whose shared node is concrete (e.g. {s p_1 ?x. s p_2 ?y}), they can be processed as two independent sub-queries without having side effects (in terms of network traffic and responding time). This is because the cardinality of the shared node (which is concrete) is not affected by any edge that connects to it. Furthermore, this observation holds if the shared node is a variable whose cardinality does not change during execution.

[7] Ψ=PSI=**P**arallel **S**ub-query **I**dentification

We generalise the above observation as follows. We say a node has a *fixed cardinality* if, during the execution of edges connecting to it, its cardinality does not change more than a certain percentage. If "removing" all fixed-cardinality nodes[8] results in disconnected sub-graphs, these sub-graphs can be optimised and executed independently and in parallel. For example, in the graph shown in figure 5.3, if both node B and C are fixed-cardinality nodes, then we have three independent sub-graphs $\{AC, AB\}, \{BC\}, \{CD, BD\}$. If only B has fixed cardinality, then the given graph cannot be further broken down[9].

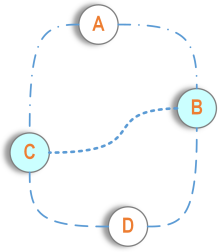


FIGURE 5.3: If B and C are fixed-cardinality nodes, there are three independent components shown by three different types of dash lines.

We propose algorithm Ψ (algorithm 1) to quickly break a connected graph into independent sub-graphs. At the beginning the algorithm creates a sub-graph for each edge (the loop at line 1). Then all nodes are scanned and sub-graphs that share a none-fixed-cardinality node are merged into a bigger one (the loop at line 4). At the end of this algorithm, all remaining sub-graphs can be processed in parallel. The time complexity of the first loop is linear to the number of edges $|E|$. The merge operation in the second loop can be done in constant time by maintaining a hash table that maps a node to the set of its connected edges. Therefore, the complexity of the second loop is linear to the number of vertices $|V|$. The complexity of algorithm Ψ (upper bound) is $O(\max(|E|, |V|))$.

In practice, concrete nodes always have fixed-cardinality. Besides, if we can know in advance that the cardinality of a variable node will probably remain the same, that node can be regarded as a fixed-cardinality node as well. For example, in {?person foaf:firstName ?frstN. ?person foaf:familyName ?fmName}, the cardinality of *?person*

[8]Since removing a node produces broken edges that have only one node, a more precise description here would be "regarding all edges that connect to a fixed-cardinality node as disconnected at this node".

[9]A more subtle case is that cardinality of both B and C are only changed by AB and AC respectively, while BC and BD have comparable cardinality at B, and BC and CD have comparable cardinality at C. That is, B and C are not fixed-cardinality nodes w.r.t all connecting edges, but they are w.r.t some edges. In this case $\{CB\}, \{CD, BD\}$ can still be executed in parallel, and we say this two components form a partial parallel group. However, identifying all partial parallel group can be costly and not worthy in practice.

Algorithm 1: Ψ(V,E)

input : A connected graph (V, E)
output: Independent sub-graphs

```
1 foreach e ∈ E do
2     sub(e) ← e;
3 end
4 foreach v ∈ V ∧ ¬ fixCard(v) do
5     merge sub-graphs containing v;
6 end
```

probably remains the same during execution, since a dataset usually contains both the first name and family name of a person. To accurately predict the invariability of a node's cardinality requires schema of properties. For instance, in the above example we need to know that both properties have the same domain, have close numbers of distinct subjects, and are closely relevant.

5.6 Optimising Queries for Parallel Execution

The optimiser of LHD-s is designed to produce parallel QEPs that explicitly indicate concurrent execution of operators. Contrary to a parallel QEP, we call it a serial QEP if contained operators are executed one after another[10]. With a cost model that considers parallel execution (such as the one presented in the previous section), it is straightforward to produce parallel QEPs with dynamic programming. However, there are cost models that do not take parallelism into account (e.g. those measure the total number of CPU instructions or network traffic) and lead to serial plans. To make LHD-s more flexible, we produce parallel plans in two steps. Given any cost model, LHD-s firstly find the optimal plans using dynamic programming as normal, then the QEPs are transformed into their parallel forms in a way that the keeps the estimated cost unchanged. This two-step method enables LHD-s to produce QEPs with either minimum responding time or network traffic.

[10]Serial QEPs can also be executed using multiple threads. For example, the optimiser of FedX generates serial plans. However, each operator in a FedX plan is executed using multiple threads. We refer to executing multiple operators in parallel as inter-operator parallelism, and using multiple threads to execute an operator as inner-operator parallelism.

5.6.1 Generating Serial Query Plans

A QEP of a SPARQL query indicates the execution order and joins of all triple patterns. To find the genuine optimal plan it is necessary to examine all permutations of possible joins (in LHD-s the operators are hash join and bind join). Dynamic programming can discard sub-optimal plans at an early stage and thus saves time. When optimising a distributed SPARQL query the access plans of triple patterns are only determined by the join operators (i.e. a dependent access plan is used only when the triple pattern is joined by a bind join). Therefore, the dynamic programming used in LHD-s starts from joining two triple patterns rather than building access plans, as shown in algorithm 2. Generally dynamic programming considers all join operators to join two sub-plans (line 5). However, there are two cases that only one join operator needs to be considered.

Algorithm 2: DP(B)

input : A BGP as a set of triple patterns B
output: The optimal QEP `OptPlan(`B`)` of the BGP

```
1 for i = 2 to |B| do
2     forall the S ⊂ B ∧ |S| = i do
3         OptPlan(S) ← ∅;
4         forall the O ⊂ S do
5             OptPlan(S) ← OptPlan(S) ∪ joinPlans(OptPlan(O), OptPlan(S\O));
              // prunePlans(optPlan(S)) is not necessary
6         end
7     end
8 end
9 return OptPlan(B)
```

First, if neither of the two sub-plans is an access plan of a triple pattern, they cannot be joined using a bind join. This is because the behaviour of executing an arbitrary operator is not clearly defined. For example, given two sub-plans $p = T_1 \bowtie T_2$ and $q = T_3 \bowtie T_4$ that are joined using a bind join $q \bowtie_B q$, it indicates that triple patterns of q should be executed with bindings of p. This is contradictory to the behaviour of executing T_3 and T_4 that is specified by q. Therefore, only hash join is used to join two join operators in LHD-s.

Second, the result of executing a triple pattern is independent of the access plan. When joining a triple pattern T with a sub-plan p, the choice of access plan can be made by comparing $cost(q \bowtie T)$ and $cost(q \bowtie_B T)$ and keeping the minimum.

Applying the above two rules can significantly reduce the number of QEPs needed to be examined (i.e. the searching space), and thus improves the performance of LHD-s' optimiser.

5.6.2 Transforming Serial Query Plans into Parallel Plans

In a serial QEP it is bind join that makes the execution order significant. If in a QEP all triple patterns are executed sequentially using independent access plans, any order of execution produces the same amount of network traffic (and the same responding time if parallelism is used). In other words, the only constraint on executing a triple pattern is whether the depending bindings of this triple pattern is available or not. For independent access plans the requirement is always met. LHD-s uses algorithm 3 to determine the execution order of all access plans, which is a dependency tree. All independent access plans are at the top level of the tree. Variables of access plans already in the trees are regarded as bound. An independent access plan is added to the current level of the tree once its dependency is met (i.e. the variable providing bindings to this access plan becomes bound).

Algorithm 3: ParlTrans(p)

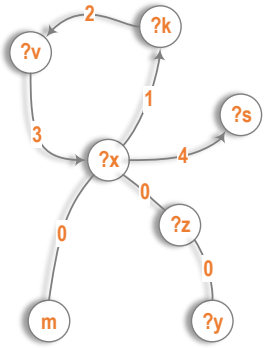
input : A QEP P
output: A dependency tree DT for the QEP

```
1  i ← 0 ;
2  bound ← ∅ ;
3  while P ≠ ∅ do
4      foreach access plan a ∈ P do
5          if depVars(a) ⊂ bound then // Dependency check
6              DT(i) ← DT(i) ∪ a ;
7              bound ← bound ∪ Vars(a) ;
8              P ← P\a ;
9          end
10     end
11     i ← i + 1 ;
12 end
13 return DT
```

A QEP produced by our algorithm can be regarded as a partial-directed (i.e. some edges are directed while others are not) graph as shown in figure 5.4. The nodes of the graph are subjects or objects and the edges are triple patterns. An undirected edge represents a plain access plan of a triple pattern, while a directed edge represents a dependent

t_1 : m p_1 ?x, 0
t_2 : ?z p_2 ?y, 0
t_3 : ?z p_3 ?x, 0
t_4 : ?x p_4 ?k, 1
t_5 : ?v p_5 ?k, 2
t_6 : ?v p_6 ?x, 3
t_7 : ?s p_7 ?x, 4

(A) Ordered triple patterns



(B) A corresponding QEP

FIGURE 5.4: An example query and its execution plan

access plan that consumes bindings from its starting node. Each edge has an execution order number, that edges with the smaller order numbers executed earlier. Execution order numbers are used to determine the execution order for edges connected to the same node, but not edges connected to different nodes (i.e. two edges of different nodes are executed in an undetermined order). The execution of the QEPs is data driven. As bindings coming from SPARQL endpoints variables become bound, and triple patterns depending on such variables are executed immediately. Thus, in LHD-s triple patterns are executed as soon as QEPs permit to exploit bandwidth.

5.7 Parallel Query Execution System

To reduce responding time, we propose a parallel execution system that adopts parallelism at two levels. The first level is that join operators are executed in parallel according to the aforementioned parallel plans. The second level includes parallel execution of each operator. For example, a bind join can be partitioned horizontally and executed using multiple threads. We decouple the execution of QEPs from the communication with (i.e. sending queries to or receiving results from) data sources. The former is controlled by the QEP executor, and the latter is managed by the communication manager. The QEP executor submits query execution tasks to the communication manager. The communication manager controls the traffic to each data source independently and determines when a task is executed according to the availability of data sources. This

design enables LHD-s to fully exploit the bandwidth and computing power of all data sources.

5.7.1 Query Plan Executor

In the QEP executor, the results of executing an access plan is regarded as a data stream. We use a quadruple $\{t, n, s, E\}$ to denote a stream which is the result of a triple pattern t; n is the level of t in the dependency tree, used as the execution order; s is the node providing binding to this stream, and E is a set of nodes where the stream goes to. For a stream corresponding to an independent access plan, s is *null* and E contains all variables of t that are used as join variables. In the case of a dependent access plan, s provides bindings and E contains the other node of t. A stream consumes bindings of s (if not empty) and pushes the results of evaluating t to the nodes in E. A node v contains a set of incoming streams In and a set of outgoing streams Out. A node joins the results of incoming streams and triggers certain outgoing streams. When the execution starts, all streams having execution order 0 start. Incoming streams that have not been processed are joined to provide intermediate result at the common node they go to. An outgoing stream starts as soon as all incoming streams having smaller execution order (i.e. the incoming streams that the outgoing stream depends on) start. Once the result of an incoming stream is consumed by an outgoing stream, the incoming stream is marked as "consumed" and will not be involved in future joins or passed to later outgoing streams. Thus a stream will not be joined twice. In case no outgoing streams exist for an incoming streams, these streams are redirected to a virtual node where stores all intermediate results that are not consumed. At the end of the execution the final result of the query is produced by joining all intermediate in the virtual node.

Figure 5.5 shows a step-by-step example of executing the QEP shown in figure 5.4. At the beginning (figure 5.5a) three streams of execution order 0, $(t_1, 0, \phi, \{?x\})$, $(t_2, 0, \phi, \{?z\})$ and $(t_3, 0, \phi, \{?x, ?z\})$ start. Since $?y$ is not used as a join variable, no streams provide data to it. The streams of t_2 and t_3 are joined and marked as consumed at node $?z$. Join result is pushed to the virtual node V since no outgoing stream exists at node $?z$. In the meantime, the streams of t_1 and t_3 are joined at $?x$ (but not marked as consumed yet). In the next step (figure 5.5b, stream $(t_4, 1, ?x, \{?k\})$ consumes the join results of streams of t_1 and t_3, both of which are marked as consumed at node $?x$, and executes

t_4 using a dependent access plan. The stream of t_7 keeps waiting since the stream of t_6, whose execution order is smaller than t_7, has not started yet. Step 3 (figure 5.5c) is similar to step 2 that executes t_5 using a dependent access plan. In step 4 (figure 5.5d) the stream of t_6 goes back into $?x$ but is not joined with any stream (since all other incoming streams of $?x$ are marked as consumed). In the final step (figure 5.5e) only the stream of t_6 is passed to the stream of t_7. The results of executing t_7 that go into node $?s$ are passed to the virtual node V. All results (as streams) at V are joined to produce the final results of a query.

5.7.2 Communication Manager

The actual execution of a query is managed by the communication manager. For each data source the communication manager maintains several worker threads that send query requests to and receive responses from the data source, and a queue that stores tasks submitted to this data source. The number of threads of each data source is set w.r.t the capability of and the connection to the data source. Once the QEP executor invokes a stream, one or several query execution requests are submitted to the communication manager. A plain access plan generates only one request. For a dependent access plan more than one request is possible since the input bindings of the dependent access plan can be partitioned into multiple segments and then executed in parallel (i.e. using horizontal partition [Kossmann, 2000] to achieve intra-operation parallelism [Hong and Stonebraker, 1993]). For example, a dependent access plan that has ten input bindings can be executed in parallel as two dependent access plans each with five input bindings each, or even ten dependent access plans each with one input binding. For each request from the QEP executor, the communication manager dispatches tasks to all relevant data sources of the triple pattern of the request. A task first goes into the task queue, waiting if all worker threads are busy, being executed otherwise. Once the task queue becomes empty, all worker threads are suspended until new tasks come in.

The main advantage of using this communication manager is to control communication to different data sources independently, and thus ensures that all data sources work at their strength without being over flooded. Furthermore, separating plan executor from communication manager enables QEP execution to proceed without waiting for actual query execution (as long as some data sources are providing result streams), and

query tasks are continuously submitted to the communication manager (to keep as many worker threads working as possible).

5.8 Summary of LHD-s

In this chapter we describe our distributed SPARQL engine called LHD-s which is designed for LD network having detailed service descriptions. LHD-s requires statistics of property partitions, including the number of total triples, distinct subjects and objects of each partition. It adopts a hybrid source selection approach (i.e. predicate-matching and *ASK* selection) and a selectivity-based responding time cost model. Given a SPARQL query, LHD-s firstly tries to break the query into sub-queries that can be optimised and executed independently. Each sub-query is optimised in two steps. Firstly, a serial QEP is generated using dynamic programming, whose complexity is reduced according to characteristics of SPARQL queries. Secondly, the serial plan is transformed into a parallel without increasing network traffic or execution time. The two-step approach allows LHD-s to use an arbitrary cost model and optimisation algorithm with parallelism. Optimised queries are executed using a highly parallel execution system. The system adopts inter-operator parallelism as well as inner-operator parallelism. Furthermore, communication with each data source is managed independently according to the bandwidth and computing power of the data source. Therefore, LHD-s can benefit from higher bandwidth without suffering overload of remote data sources.

5.9 Implementation of LHD-s

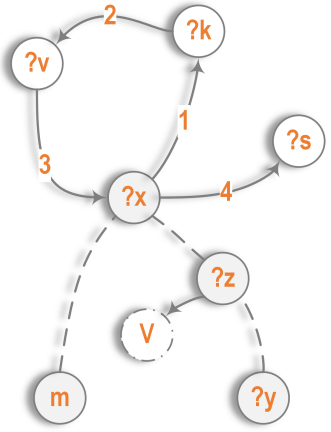
The execution system of LHD-s is built using pipelined parallelism such as Double-pipelined Hash Join [Raschid and Su, 1986] and XJoin [Urhan and Franklin, 1999]. Instead of using two hash tables like in a Double-pipelined Hash Join, we maintain multiple hash tables at a node to enable joining more than two streams simultaneously. A result coming from one stream is stored in the hash table of this stream, and at the same time probed against the hash tables of other streams. For example, at a time three streams a, b and c are joined at a node $?x$, and three hash tables H_a, H_b and H_c are maintained respectively. Once a result comes from a, it is stored to the hash table H_a under the key of the value of $?x$, and probed against H_b and H_c on the same value of

$?x$. A join result is produced as soon as matching records are found in both H_b and H_c, and given to outgoing streams that consume the result. This multiple hash join enables a node to execute several incoming streams as well as outgoing streams in parallel. The execution will not be delayed unless all data sources stop providing results.

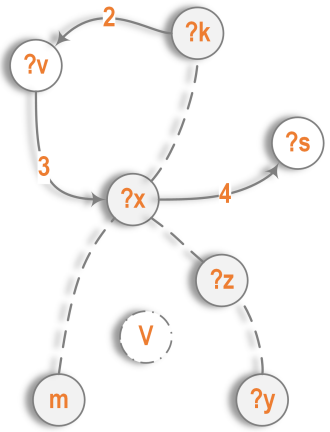
When executing a dependent access plan there could be duplicated values of the depended variable (e.g. considering two input bindings $(?x \rightarrow x_1, ?y \rightarrow y_1), (?x \rightarrow x_1, ?y \rightarrow y_2)$ for triple pattern $\{?x\ p\ ?z\}$, only one value $(?x \rightarrow x_1)$ is required by the dependent access plan of the triple pattern). To eliminate unnecessary network traffic, we propose the Hash Bind Join (HBJ) operator, that partitions the input bindings using a hash table on the values of the depending variable $(?x)$. Therefore we only use distinct values to execute a dependent access plan, and the returned results of a specific input value are joined with bindings of the same value in the hash table. When only one binding is given for a dependent access plan, variables in the triple pattern of this access plan are replaced by values of the given binding (i.e. the implementation of bind join in DARQ and DSP). Otherwise the input bindings are attached as inline data using the *VALUES*[11] syntax in the dependent access plan. For example, to execute $\{?x\ p\ ?z\}$ with input bindings $(?x \rightarrow x_1, ?y \rightarrow y_1), (?x \rightarrow x_1, ?y \rightarrow y_2)$, firstly a hash table $x_1 \rightarrow \{(?x \rightarrow x_1, ?y \rightarrow y_1), (?x \rightarrow x_1, ?y \rightarrow y_2)\}$ is built. Then $\{x_1\ p\ ?z\}$ is evaluated against relevant data sources. The results $(?z \rightarrow z_1), (?z \rightarrow z_2)$ are joined with $(?x \rightarrow x_1, ?y \rightarrow y_1), (?x \rightarrow x_1, ?y \rightarrow y_2)$ to produce the complete results.

The communication manager maintains a thread pool and a task pool for each data source. The number of thread in each thread pool is set to a number (as large as possible) that is lower than the maximum allowed concurrent connections to the corresponding data source. The execution tasks to a data source are stored in its task poll. An idle thread is invoked to executed a task if the task pool is not empty, otherwise all threads are paused.

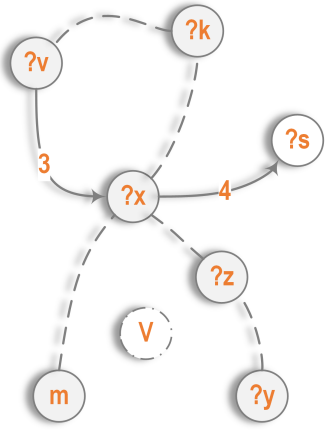
[11] `http://www.w3.org/TR/sparql11-query/#inline-data`



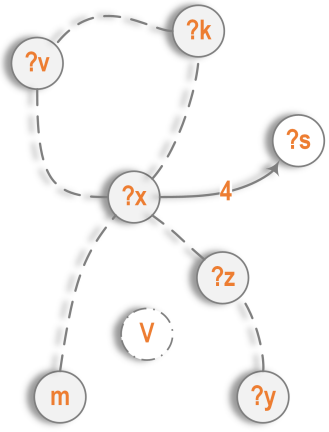
(A) Query execution step 1. The execution starts with three streams of order 0, represented by dash lines. Joined result of t_2 and t_3 is pushed to the virtual node V for later use. Stream t_1 and t_2 are joined at $?x$ and wait to be consumed by streams of order 1.



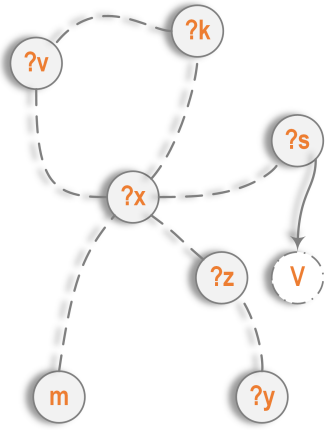
(B) Query execution step 2. Stream t_4 which has execution order 1 starts since all streams having smaller execution order have started. It consumes the join result of t_1 and t_2 and pushes its result to $?k$. Stream t_7 which has an execution order 4 remains halting since the stream having execution order 3 has not started yet.



(C) Query execution step 3. Stream t_5 starts execution in the same manner as stream t_4. It consumes the result of t_4 and pushes result to $?v$.



(D) Query execution step 4. Stream t_6 goes back to $?x$. Since all other incoming streams of $?x$ have been consumed, t_6 will not be joined with them. Now all incoming streams at $?x$ started and t_7 is ready to start.



(E) Query execution last step. Stream t_7 starts since all incoming streams at $?x$ whose execution orders are smaller than 4 have started. The result of t_7 is pushed to the virtual node V, at where it is joined with the join result of t_2 and t_3 to produce the final result.

FIGURE 5.5: Execution of a QEP

Chapter 6

Evaluating LHD-s

To demonstrate the performance of the combination of techniques for environments having detailed VoID descriptions, we evaluate LHD-s and compare it with existing distributed SPARQL engines. In particular, the cost model is evaluated in a calculation-based experiment. Meanwhile, it is difficult to either comprehensively evaluate the optimisation algorithm and the query execution system using only theoretical analysis, or to isolate their performance. As a result, the performance of the optimisation algorithm and the query execution system are evaluated together using the evaluation framework presented in chapter 4. Engines used for comparison are FedX and SPLENDID. FedX adopts a straightforward heuristic-based optimisation approach and a sophisticated parallel execution system. SPLENDID emphasises on query optimisation while using a pipeline-based single thread execution system. Their approaches are closely related to what we developed in LHD-s, and are promising to be good references. Other engines are considered less appropriate for various reasons. For example, DARQ also provides a set of well designed techniques that are related to our approach. However, it is not up to date and several evaluations Schwarte et al. [2011], Wang et al. [2011] indicate that its efficiency is not as good as recent engines. Another recent query engine, ANAPSID Acosta et al. [2011], focuses on adaptive query processing techniques to cope with unstable networks. Also, insufficient details are given about its query optimisation and query execution (except the adaptive processing part). Therefore ANAPSID is not used as a reference.

6.1 Evaluating Cost Models

The core of cost models of LHD-s, FedX and SPLENDID is to estimate the cardinality (the number of matching triples) of a single triple pattern or joins of triple patterns. To measure these cost models' accuracy, we compare the estimated cardinality of joins to the real cardinality that are obtained by executing the joins. For single triple pattern, the actual and estimated cardinality are equal if both the subject and object are variables, in which case the number of triples matching the predicate equals the number of triple falling into this predicate partition. In case the subject or object is concrete, the actual cardinality mainly depends on the specific value of the concrete subject or object. Therefore, to have the accurate cardinality of triple patterns with only one variable, we have to count over triples for every pair of subject-predicate or object-predicate. At the same time, existing cost models (in DARQ, SPLENDID and LHD-s) adopt quite similar methods to estimate a triple pattern with one variable due to limited statistics. As a result, we do not compare cost models on estimating a single triple pattern. We also exclude joins of an arbitrary number of triple patterns (i.e. n-nary joins). Because obtaining the actual cardinality of n-nary joins requires executing all permutations of triple patterns in them. In this thesis, we only evaluate cost models on estimating binary joins whose triple patterns have only concrete predicates (e.g. $\{?s\ p_1\ ?o_1,\ ?s\ p_2\ ?o_2\}$). We will perform a more comprehensive test on cost models in the future.

6.1.1 Evaluation Method

To collect the actual cardinality of an arbitrary binary join of the aforementioned type, we generate all possible joins and execute them against the evaluation datasets. Since predicates of different queries can be irrelevant, we only join predicates of the same query. For every query in the query set of the evaluation framework, we collect the distinct predicates. For every two predicates p_1 and p_2 we generate a query in the form of $\{?x\ p_1\ ?o_1,\ ?x\ p_2\ ?o_2\}$ (SS join) and $\{?s_1\ p_1\ ?x,\ ?x\ p_2\ ?o_2\}$ (OS join). Taking Query 1 (in table A.1) as an example, the SS and OS joins of its four predicates (*bsbm:productFeature* (pF), *bsbm:productPropertyNumeric1* (pPN1), *rdfs:label* (lbl) and *rdf:type* (a)) are shown in table 6.1. SS joins are listed in the top-right half of the table, and OS joins are in the bottom-left part. Joins generated based on each query are merged and duplications are eliminated. Due to the similarity of SS and OS joins, we only give the complete list

of SS joins (123 joins in total) in appendix A.2. With the actual cardinality of triple patterns we calculate the join selectivity of each join.

TABLE 6.1: Possible SS & OS joins of Query 1. There are four triple patterns in Query 1. The upper right part of the table contains all SS joins of arbitrary two different triple patterns. The lower left part contains all OS joins of arbitrary two different triple patterns.

	pF	**pPN1**	**lbl**	**a**
pF		?x pF ?o1. ?x pPN1 ?o2.	?x pF ?o1. ?x lbl ?o2.	?x pF ?o1. ?x a ?o2.
pPN1	?s1 pPN1 ?x. ?x pF ?o2.		?x pPN1 ?o1. ?x lbl ?o2.	?x pPN1 ?o1. ?x a ?o2.
lbl	?s1 lbl ?x. ?x pF ?o2.	?s1 lbl ?x. ?x pPN1 ?o2.		?x lbl ?o1. ?x a ?o2.
a	?s1 a ?x. ?x pF ?o2.	?s1 a ?x. ?x pPN1 ?o2.	?x a ?x. ?x lbl ?o2.	

We estimate each SS and OS join using cost models of FedX, SPLENDID and LHD-s respectively and calculate the corresponding join selectivity. For clarity we describe the behaviour of the three cost models on these joins. FedX counts the number of variables of each triple pattern and all SS (or OS) joins are considered equal. Therefore we say that FedX gives a constant cardinality for the same type of joins. We enlarge FedX's estimations by 6 orders of magnitude to make estimations of all engines have a close order of magnitude. SPLENDID uses equation $\prod sel.s \cdot card(T_{unbound})$ to estimate cardinality of SS joins. For OS join it uses equation $card(T_1) \cdot card(T_2) \cdot sel(T_1 \bowtie T_2)$, where $sel(T_1 \bowtie T_2)$ is the average selectivity of the join variable. In the original paper it is not clear what the average selectivity of a variable refers to. However from the source code of SPLENDID it is the arithmetic average of the selectivity of the join variable (i.e. $sel(SO) = (sel.s(p_1)+sel.s(p_2))/2$. LHD-s uses equation $card(T_1) \cdot card(T_2) \cdot sel(T_1 \bowtie T_2)$ to estimate both SS and OS joins, where the join selectivity is given by equation 5.6.

It is not necessary that good cost models have to produce cardinality estimations close to actual cardinality. In query optimisation it is the rank of joins by cardinality that matters. Query optimisation will produce the same QEP, if for any two joins A and B in a query, the rank (whether $card(A)$ is larger or less than $card(B)$) produced by using A and B's estimated cardinality is the same as the ranking using their actual cardinality. Therefore we introduce the concept of *ranking accuracy*, which is the percentage of

correct rank of arbitrary two joins based on estimated cardinality, to measure the quality of cost models. To calculate the ranking accuracy we go through all pairs of joins. We divide join pairs into different groups according to their *distance*, that is, in the group of distance n, all pairs contain the ith and the $i+n$th joins. Any pair of triple patterns belongs to a group. We check whether their estimated rank according to a cost model is the same as actual rank and calculate the ranking accuracy of the cost model.

6.1.2 Results and Analysis

Join cardinality produced by the three cost models are compared with the actual cardinality in figure 6.1 and 6.2. Since many of the OS joins have 0 result, we present cardinality of none-zero OS joins separately in figure 6.3.

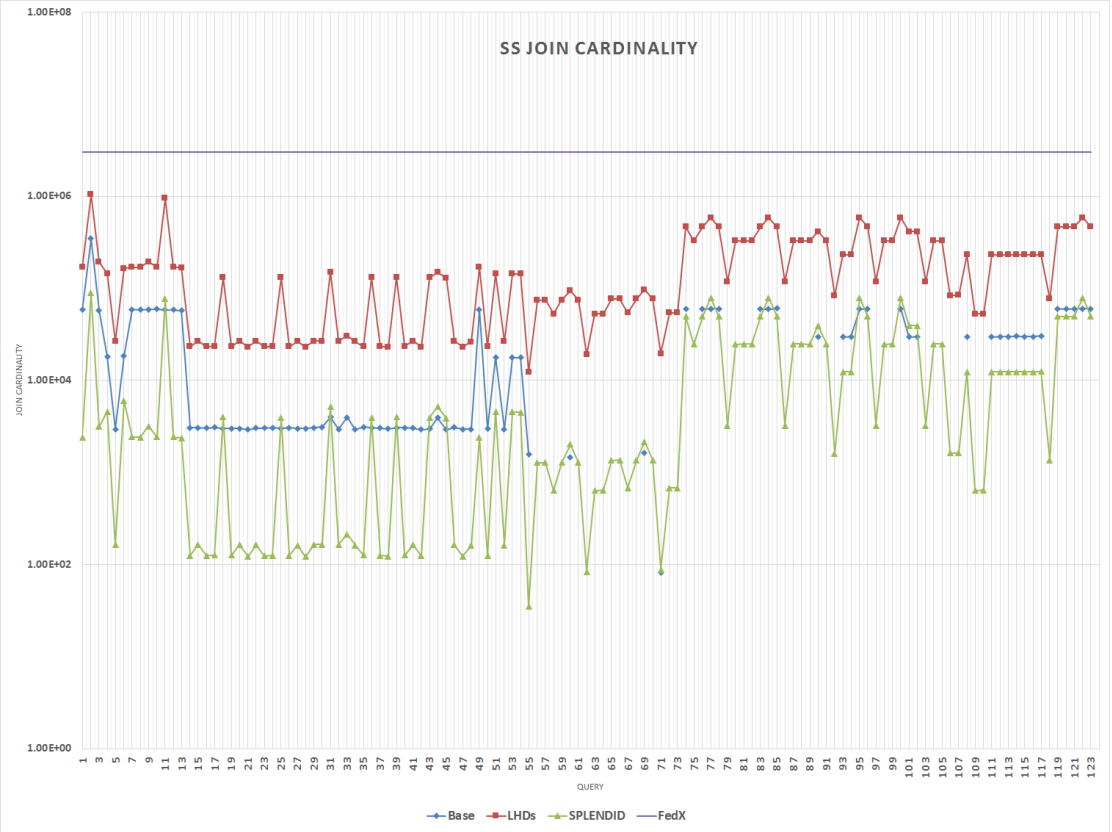


FIGURE 6.1: Base stands for the actual cardinality of joins. Gap indicates cardinality of 0. Lines are for visual aid only.

Figure 6.1 and 6.2 shows that none of the cost models gives accurate estimations (all estimations are several orders of magnitude away from the actual values). Therefore, we focus on examining the trend of these estimations instead. In the aforementioned figures joins from the same query are shown as points next to each other. It is shown

in figure 6.1 that cardinality produced by LHD-s and SPLENDID have a trend close to the actual cardinality of SS joins. The ranking accuracy of each cost model is produced by joins of up to a distance of 12 (i.e. the maximum possible distance since the largest BPGs in this case have 12 triple patterns), as presented in table 6.2.

TABLE 6.2: Comparison of ranking accuracy on SS joins

DIST.	**LHD-s**	**SPLD.**	**FedX**
1	72	70	41
2	74	72	44
3	75	72	40
4	66	63	44
5	71	67	42
6	71	68	40
7	73	72	39
8	71	70	42
9	78	76	42
10	68	66	48
11	65	64	46
12	72	71	44

It is clearly shown in table 6.2 that LHD-s has a slight advantage over SPLENDID on joins of all presented distance. This is because that LHD-s produces a more accurate join selectivity of SS joins than SPLENDID does (which is constant $sel.s$). In the meantime, FedX's comparison accuracy is lower than 50%, which indicates it probably gives an incorrect comparison for an arbitrary two SS joins (having only concrete predicates). It is worth noticing that FedX has several heuristics that are used to compare triple patterns having concrete subjects or objects, and may show better performance for types of joins that are not considered here.

On OS joins, even on none-zero OS joins, all engines fail to resemble the actual trend. This implies that the main factor affecting cardinality of OS joins is not captured by any of the engines. Due to the frequent occurrences of 0 in actual cardinalities, it may be possible to identify such joins through analysis of the predicate schema. For instance, if the range of the first predicate is disjoint with the domain of the second predicate, we will conclude that the join is empty. Following this line, methods discussed in Akar et al. [2012] may also be used to further refine estimation of OS joins, and it is part of the future work.

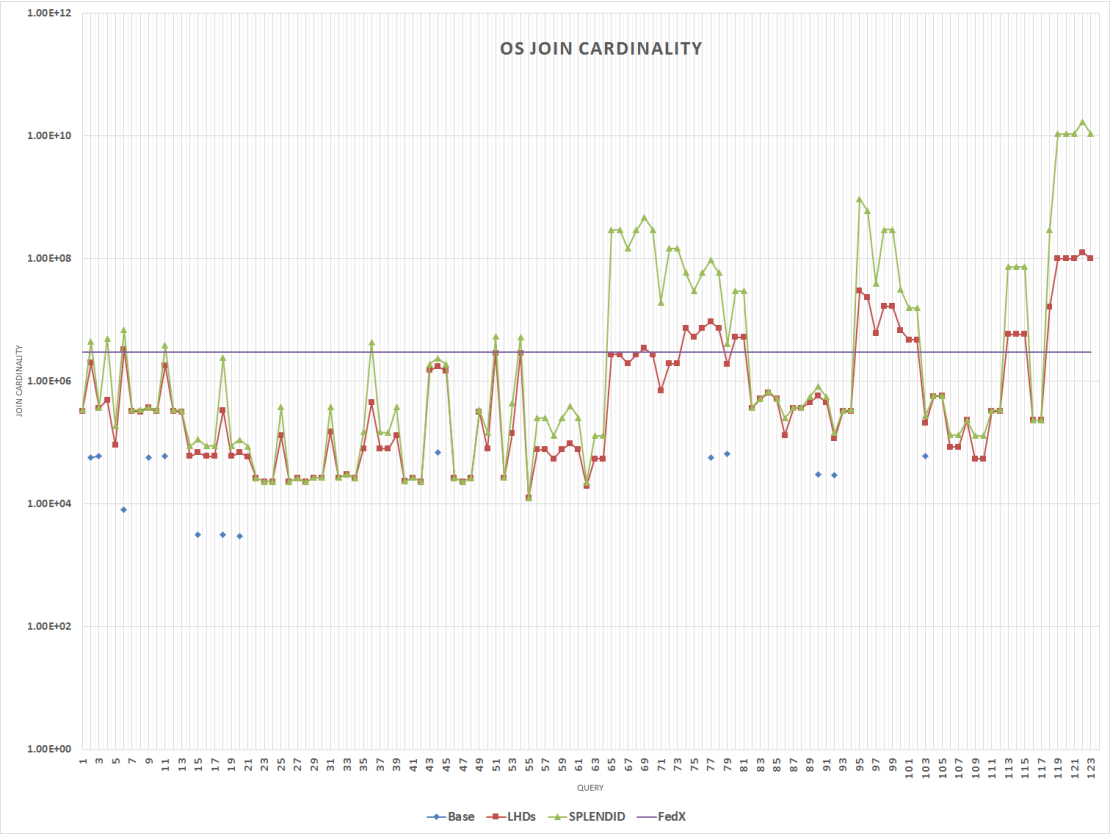


FIGURE 6.2: Base stands for the actual cardinality of joins. A gap indicates cardinality of 0. Lines are for visual aid only.

6.2 Evaluating the Optimisation Algorithm and the Execution System

The effectiveness of optimisation algorithms and the efficiency of execution systems are closely related actual queries. Instead of a calculation-based experiment, we evaluate LHD-s using the evaluation framework and measure the optimisation algorithm and the execution system indirectly via the overall query responding time and network traffic.

6.2.1 Experiment Settings

The experiment is set up using the framework presented in chapter 4. We prepare about 70 million RDF triples, which is in line with existing approaches [Quilitz, 2008, Schmidt et al., 2011]. These triples are distributed to 20 SPARQL endpoints, which is largest number of endpoints we can host, and more than existing approaches since we tend to

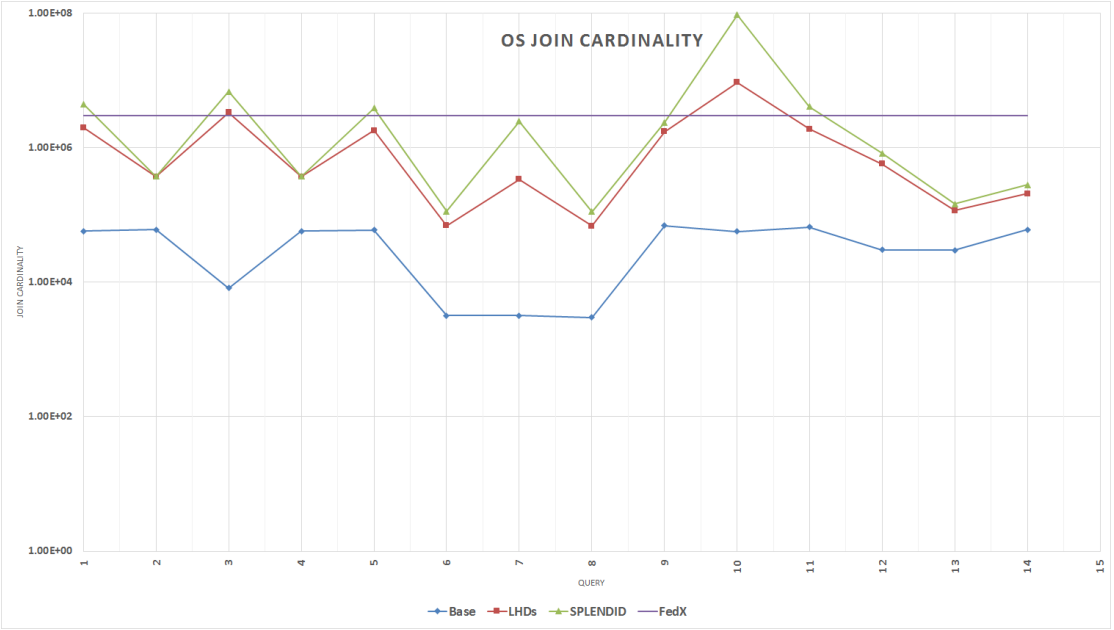


FIGURE 6.3: Base stands for the actual cardinality of joins. A gap indicates cardinality of 0. Lines are for visual aid only.

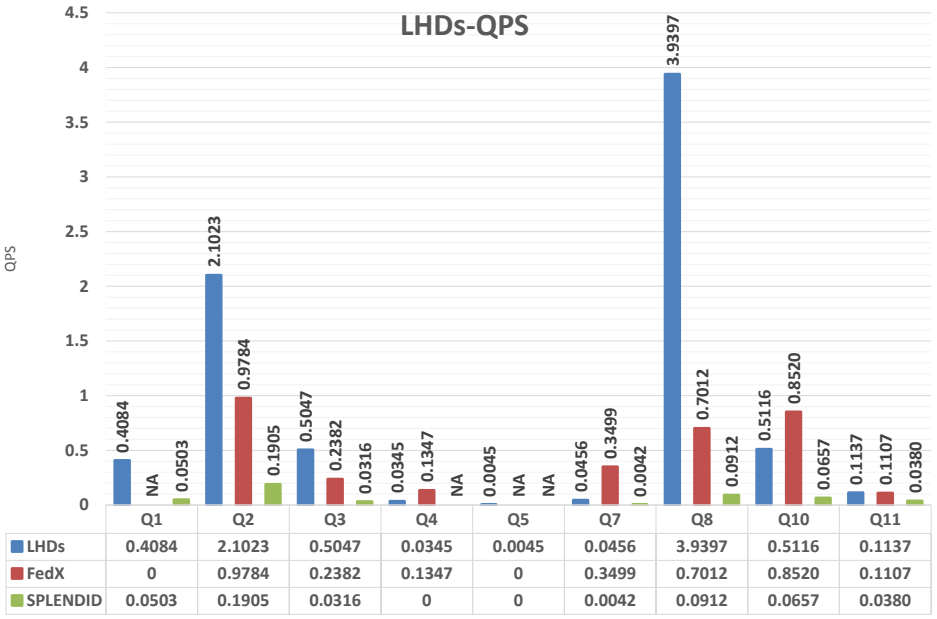
perform evaluation on a larger scale. The backend of the endpoints are Sesame 2.6[1] and Apache Tomcat 6[2]. Every two endpoints are hosted in a remote virtual machine having 2.5 GB memory. The number of triples in each virtual machine is balanced to prevent overrun of certain machines (e.g. the endpoint having maximum triples and the one having minimum triples are hosted in the same virtual machine). Each distributed SPARQL engine under testing (e.g. LHD-s) is run on a machine equipped with an Intel Xeon W3520 2.67 GHz processor and 12 GB memory.

For each query presented in chapter 4, we perform 5 warm-up runs, and 20 test runs, which enable all engines to have stable performance. The number of warm-up and test runs are determined according to existing benchmarks [Bizer and Schultz, 2009, Schmidt et al., 2011] as well as our own experiences. For each run a fresh instance of query engine is used (i.e. each run can be regarded as standalone query processing). An engine is considered incapable of finishing a query if it either 1) takes more than 5 minutes to execute the query, or 2) keeps running into execution issues (such as not enough memory or overrunning SPARQL endpoints) in three tries.

[1] `http://sourceforge.net/projects/sesame/files/Sesame%202/2.6.0/`
[2] `http://tomcat.apache.org/download-60.cgi`

6.2.2 Results and Analysis

The QPS, incoming network traffic, outgoing network traffic and transmission rate are shown in figure 6.4, 6.5, 6.6 and 6.7 respectively. In these figures 0 or NA stand for time out queries.



	Q1	Q2	Q3	Q4	Q5	Q7	Q8	Q10	Q11
LHDs	0.4084	2.1023	0.5047	0.0345	0.0045	0.0456	3.9397	0.5116	0.1137
FedX	0	0.9784	0.2382	0.1347	0	0.3499	0.7012	0.8520	0.1107
SPLENDID	0.0503	0.1905	0.0316	0	0	0.0042	0.0912	0.0657	0.0380

FIGURE 6.4: QPS of LHD-s

Figure 6.4 shows that LHD-s is faster than FedX on most queries (except Q4, Q7 and Q10), and considerably faster than SPLENDID on all queries. In addition, LHD-s is the only engine that successfully finishes all queries. FedX has time out on Q1 and Q5, while SPLENDID has time out on Q4 and Q5. The time out is caused by the large amount of intermediate results which is a result of low quality QEPs, especially for Q5.

Due to the design of BSBM, all assessment queries should be regarded as equally common which property is inherited by the DSEF queries. In addition, all engines under testing do not optimise for specific types of queries. Better performance on most queries suggests higher probability of better performance on arbitrary queries. The higher performance of LHD-s is primarily due to its parallel execution system, as we will demonstrate later.

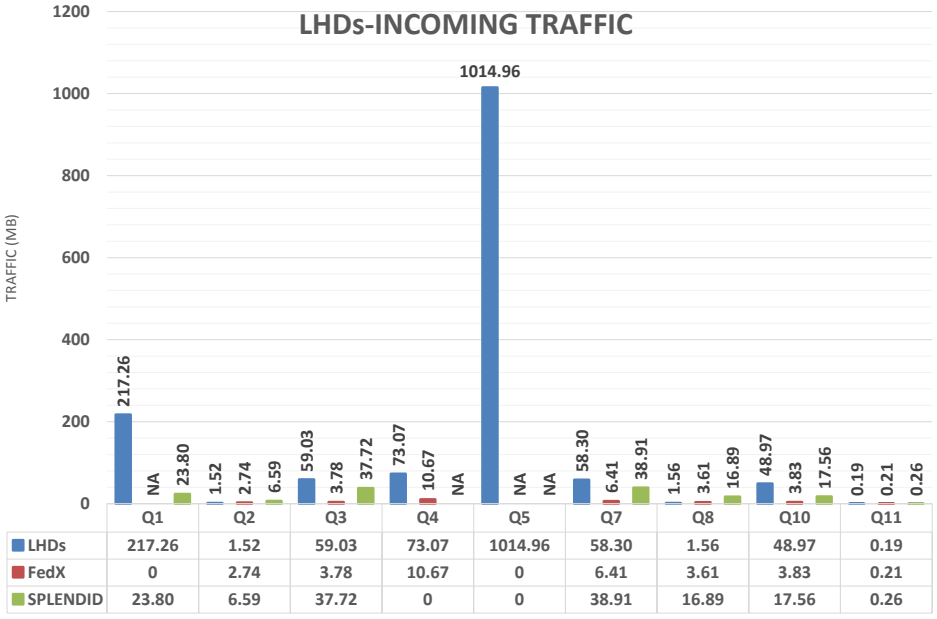


FIGURE 6.5: Incoming traffic of LHD-s

LHD-s produces the most network traffic among the three, as shown in Figure 6.5 and 6.6. One reason is that LHD-s optimises for minimum responding time rather than minimum traffic. It is worth noticing that the extra network traffic is primarily due to the incoming traffic, which implies that more hash joins are used instead of bind joins. It can be a deliberate decision of the cost model to reduce responding time, or due to incorrect cost estimation. A noticeable spike is shown in both Figure 6.5 and 6.6 on Q5. This is because Q5 generates huge amount of intermediate results which is difficult to reduce. The huge amount of intermediate results also leads to the time out of SPLENDID and FedX.

In section 5.7 we mentioned the side effects of the inter-operator parallelism (i.e. executing multiple operators in parallel) of LHD-s. That is, if operators that execute triple patterns are executed sequentially, bindings of a certain variable are likely to be reduced as more triple patterns being executed. When operators are executed in parallel, the same set of bindings may be used multiple times before reduction. This side effect can potentially increase outgoing traffic. However, Figure 6.6 shows no sign of outgoing traffic increase and indicates the side effect is not significant for the tested queries.

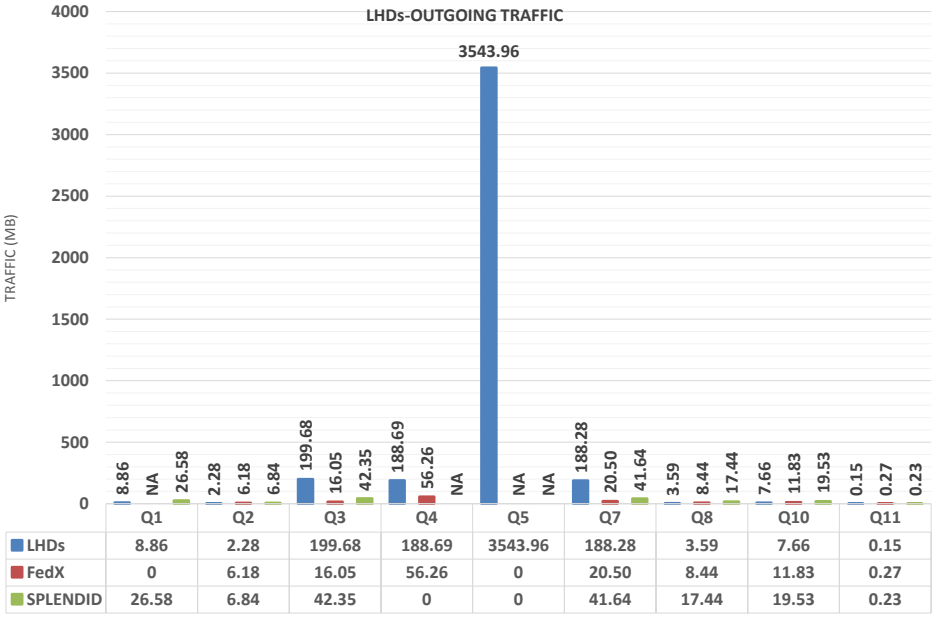


FIGURE 6.6: Outgoing traffic of LHD-s

While both optimising for minimum network traffic, both figure 6.5 and 6.6 suggest that FedX has a certain advantage over SPLENDID, except on Q1. Since dynamic programming, which is adopted in SPLENDID, provides optimal plans w.r.t the cost model, SPLENDID's higher traffic can only result from inaccuracy of its cost model or statistics. Since LHD-s and SPLENDID use similar methods to estimate cardinality, inaccuracy of cost estimation is very likely in LHD-s as well. It could be that existing cost models do not sufficiently exploit statistics of data sources. On the other hand, it is possible that VoID cannot satisfy the requirement of more sophisticated cost models. By extending the evaluation of cost models (section 6.1) to include arbitrary joins it is possible to determine whether cost models of LHD-s and SPLENDID have acceptable accuracy on n-nary joins. Furthermore, comparing QEPs generated by the three engines with the actual optimal plans (identified by experiment) will enable us to precisely measure the quality of each engine's query optimisation. It will also provide insight of which triple patterns are executed at an inappropriate time. These evaluations are in our future plan.

The efficiency of LHD-s is primarily due to its parallel execution system, which increases

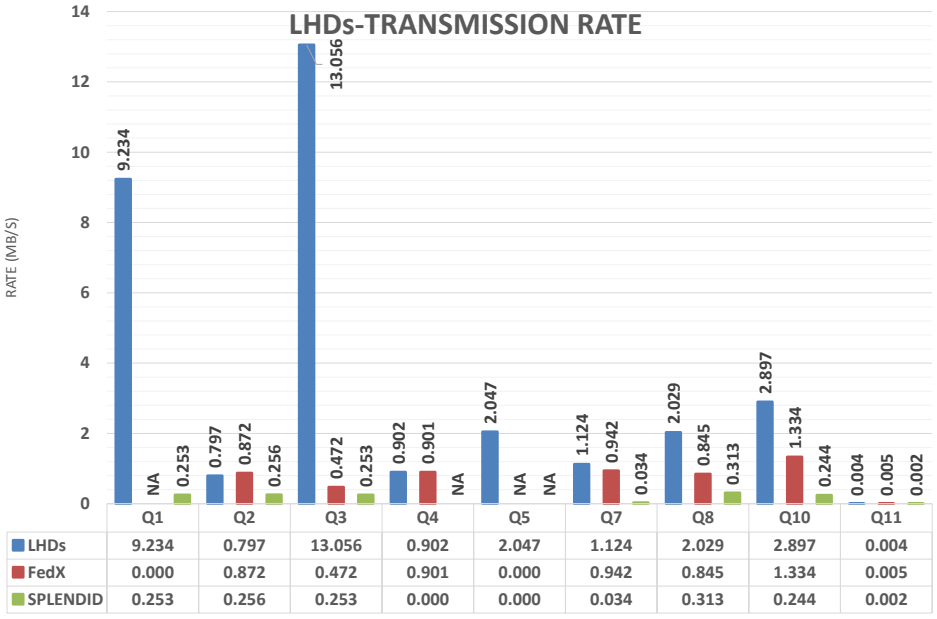


FIGURE 6.7: Average transmission rate of LHD-s

the transmission rate, as clearly shown in figure 6.7. In particular, the efficiency primarily comes from the inter-operator parallelism of LHD-s. We can see that the network traffic of the three engines on Q11 are comparable, since Q11 consists of single-triple BGPs where no optimisation takes effect. The differences between the network traffic of the three engines are primarily the results of querying overhead. For example, both FedX and SPLENDID send *ASK* queries to select relevant data sources. In addition, no inter-operator parallelism is used in LHD-s since there is only one pattern in each BGP. Therefore, the transmission rate on Q11 is merely a result of inner-operator parallelism (i.e. executing an operator with multiple threads). Since LHD-s and FedX have a close transmission rate (and is higher than SPLENDID), it implies that their inner-operator parallelism is equally effective at a low level of traffic amount[3]. Therefore, we conclude that the advantage of LHD-s is essentially brought by the inter-operator parallelism. The inter-operator parallelism of LHD-s is enabled by parallel QEPs (generated using algorithm 3), and reinforced by the mechanism that maintains an independent thread pool for each data source (i.e. potentially more threads are usable).

[3]With a large amount of network traffic FedX can suffer in that it does not maintain independent thread pools for data sources. However, we cannot confirm that due to lack of appropriate testing queries.

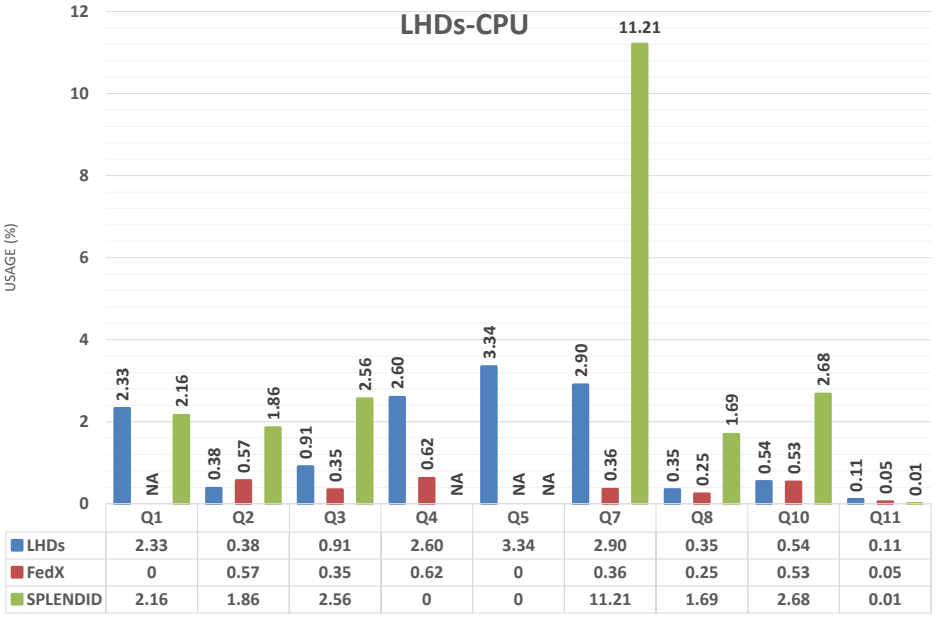


FIGURE 6.8: CPU usage of LHD-s

We also provide the average CPU and memory usage of the three engines in figure 6.8 and 6.9. However, the purpose of providing such information is only to demonstrate that these engines require reasonable amounts of system resources. It is likely that the system resources consumption of the three engines can be reduced with better engineering. All three engines have low CPU usage. One possible reason for higher CPU usage of LHD-s and SPLENDID than FedX is that the former two adopt dynamic programming which is more complex than FedX's heuristic-based optimisation. The memory consumption of LHD-s and SPLENDID is also higher than FedX. This is primarily due to FedX materialised a limited number of intermediate results[4] at one time, while the other two materialised all available intermediate results (using normal lists). Therefore, memory consumption of LHD-s and SPLENDID is likely to be reduced if less intermediate results are materialised at one time. Moreover, LHD-s potentially maintains more threads than the others and therefore consumes more memory.

[4]This is implemented using a data structure called *queue* in Java. It is essentially a special list that blocks input if a certain number of entries exist in the list.

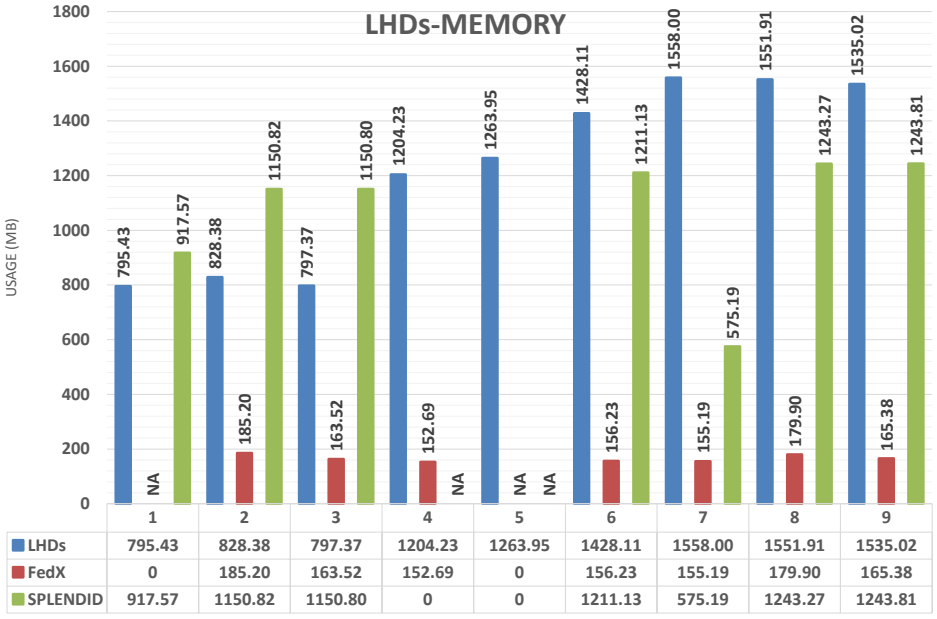


FIGURE 6.9: Memory usage of LHD-s

6.3 Evaluation Summary

In this chapter we evaluate a combination of techniques that include a VoID-based response time cost model, a dynamic-programming-based optimisation algorithm, an algorithm called Ψ to increase parallelism, and a parallel execution system. These techniques are deployed in a distributed SPARQL engine named LHD-s. The efficiency of LHD-s essentially benefits from its parallel execution system. In the meantime, such gain comes at the cost of higher network traffic. Furthermore, it is shown that on joins of two triple patterns, VoID-based cost models are more accurate than heuristics, and the cost model used in LHD-s has advantage over other VoID-based cost models. In the meantime, none of these VoID-based cost models are not sufficiently accurate, and potentially lead to sub-optimal QEPs. It may due to either that existing VoID-based cost models do not fully exploit VoID statistics, or that VoID is not able to provide enough statistics for producing accurate estimations. This will be further investigated in our future work with more sophisticated experiments.

Chapter 7

Optimising Queries with the Presence of Co-reference

LINKED Data are published by a large amount of independent publishers and little coordination exists among them. In the LD cloud a URI uniquely identifies one resource, meanwhile, a resource can have more than one URIs. Publishers are encouraged to reuse existing URIs to increase interoperability [Heath and Bizer, 2011, Hyland et al., 2013]. On the other hand, they are free to create their own URIs when publishing LD. On class[1] level, several vocabularies covering common domains, such as Friend of a Friend (FOAF)[2], and Dublin Core Metadata Initiative (DCMI)[3], are shared in many datasets. On the instance level, however, poor agreement is made on reusing URIs [Hogan et al., 2007]. For example, 23 different URIs are found referring to the person Tim Berners-Lee out of 1.118 g statements [Hogan et al., 2012]. This phenomenon, that multiple URIs refer to the same resource, is known as co-reference. Co-reference exists in several fields such as linguistics and knowledge management. Its existence is due to "inherently distributed and disparate nature of the information" [Glaser et al., 2007]. Furthermore, the information carried by an URI may depend on the context in which the URI is used. It is unlikely that a single URI is accepted in all specific datasets in the LD cloud. One solution is provided by the OWL [Carroll et al., 2012] vocabulary, which provides for co-referent URIs to be linked using the "*owl:sameAs*" property. Much work have been done to resolve co-referent URIs for

[1] A class is regarded as a common name of a set of things.
[2] `http://www.foaf-project.org/`
[3] `http://www.dublincore.org/documents/dcmi-terms/`

LD [Hogan et al., 2007, Jaffri et al., 2008, Glaser et al., 2009, Hu et al., 2011a, Umbrich et al., 2012].

In this chapter we do not explore co-reference resolution. Rather, we only examine explicit co-reference (i.e. that is presented as *owl:sameAs* statements) and investigate query optimisation having co-reference taken into account. Co-reference will not disappear as LD evolves [Glaser et al., 2009]. Considering and coping with co-reference would help further exploit the wealth of LD.

7.1 Challenges of Optimising Queries having Co-reference

For convenience of later discussions, we firstly extend the notion of co-reference to queries. We say two queries are co-referent, or one is the other's co-reference, if one query is obtainable from replacing some URIs in the other query, with the co-referent URIs of those URIs. It is worth mentioning that a query's co-reference can be high even if the concrete URIs in it only have a small amount of co-reference. Including co-reference in SPARQL queries is about combining results that match any co-referent version of the original queries. To the best of our knowledge, the OpenLink Virtuoso is the only distributed engine that provides support of co-reference in a recent release[4]. However, it focuses on co-reference resolution rather than query optimisation. Due to the lack of existing solutions, users who want to have complete results of a query having co-reference have to i) retrieve co-referent URIs for each URI in this query, ii) issue a new query w.r.t each combination of co-referent URIs (i.e. executing the original query as well as its co-referent slinging), and iii) combine results of all these queries. The total number of involved queries equals to the number of the Cartesian product of co-referent URIs of each URI in the original query. This naïve approach can imply significant overhead and poor performance when applied in the LD cloud. In the remain part of this chapter this approach is referred to as the baseline approach to which our techniques are compared.

Furthermore, introducing co-reference into distributed SPARQL queries potentially increases the sizes of results and alters statistics in a nondeterministic manner. In chapter

[4]The support of co-reference first occurred in version 6.1.5 of OpenLink Virtuoso, which can be found at `http://freecode.com/projects/oplvirt/releases/342712`.

5 we show initial evidence that existing VoID-based cost models are not sufficiently accurate, even with detailed VoID statistics. With co-reference this issue becomes more severe since statistics of co-reference are not known. Due to the large scale of LD, it is expensive, and sometimes impossible, to directly collect statistics of co-reference.

Consequently, a third challenge arises. Most existing distributed SPARQL engines follow the static optimisation approach (including DARQ, SPLENDID, FedX and LHD-s). Since accurate cost estimation is not guaranteed, static optimisation can be less effective. Alternatively, taking advantage of run-time statistics and re-optimising queries adaptively during query execution can be promising to improve query performance.

Besides, it has been shown that the semantic of *owl:sameAs* is not strictly followed in the real world LD. According to OWL, *owl:sameAs* is symmetric and transitive. However, this is only true when the URIs linked by *owl:sameAs* refer to exactly the same resource. In practice, co-referent URIs usually refer to similar resources [Halpin and Hayes, 2010] or different aspects of a resource [Glaser et al., 2007]. Also, the equivalence of URIs is usually context-dependent [Jaffri et al., 2008]. The above facts imply that, when querying the LD, we have to distinguish co-referent URIs from different datasets and cannot take advantage of the transitivity of *owl:sameAs*.

The final issue relates to the difficulty of distributed inferencing.Taking transitivity of *owl:sameAs* as an example, if two statements {a owl:sameAs b} and {b owl:sameAs c} are contained in different datasets, it is difficult to know the equivalence of a and c, unless both statements are merged locally. The same issue applies on the symmetric property of *owl:sameAs*. The symmetric breaks if an *owl:sameAs* statement is not contained reciprocally by both the owners of the subject and the object. As a result we may get different results with co-referent queries. These issues are more related to co-reference resolution rather than querying LD. Meanwhile, the above issues become easier to be addressed by a third-party co-reference services such as sameas.org.

In summary, processing queries having co-reference requires an efficient way to integrate results from co-referent queries. Meanwhile, co-reference increases result sizes and aggravates the difficulty of cost estimation. In the remain part of this chapter, we explore optimisation techniques for distributed queries in environments with co-reference by means of LHD-d, which is a distributed SPARQL engine that we developed for that purpose.

7.2 Overview of Optimisation Techniques in Environments with Co-reference

To improve the efficiency of query processing in environments with co-reference, we propose novel techniques to address the unique challenges described above. It is assumed that co-reference statements are explicitly provided by RDF datasets, and our techniques focus on improving query efficiency rather than co-reference resolution.

First, we propose a model called Virtual Graph to integrate co-reference. Using Virtual Graph, queries having co-reference are transformed into normal queries that can be optimised and executed using existing approaches.

Second, different from the static optimisation approach used in environments having detailed statistics, here we optimise queries during query execution (i.e. dynamic optimisation). This enables LHD-d to take advantage of runtime statistics such as the actual number of results of triple patterns. Consequently, LHD-d uses a MST-based algorithm to incrementally construct the optimal QEPs.

In addition, we further exploit the Ψ (algorithm 1) with the help of runtime statistics. Query optimisation and execution are interleaved and highly parallel.

Besides the above techniques that are especially tailored for query processing with co-reference, several techniques that have been successfully tested with LHD-s are adopted as well: VoID is used as complementary statistics for choosing the first step in the dynamic optimisation (when the runtime statistics are not available yet); predicate matching is used for source selection; and the parallel execution system is used to increase transmission rate.

We deployed the aforementioned techniques in LHD-d, which is shown in figure 7.1. Details of each component is given in the following sections.

7.3 Addressing Co-reference using Virtual Graph

Given a SPARQL query having URIs with co-reference, the query itself as well as its co-referent queries have to be executed to get comprehensive results. For example, given a triple pattern {?x foaf:knows p_0} and a co-reference statement {p_0 owl:sameAs p_1},

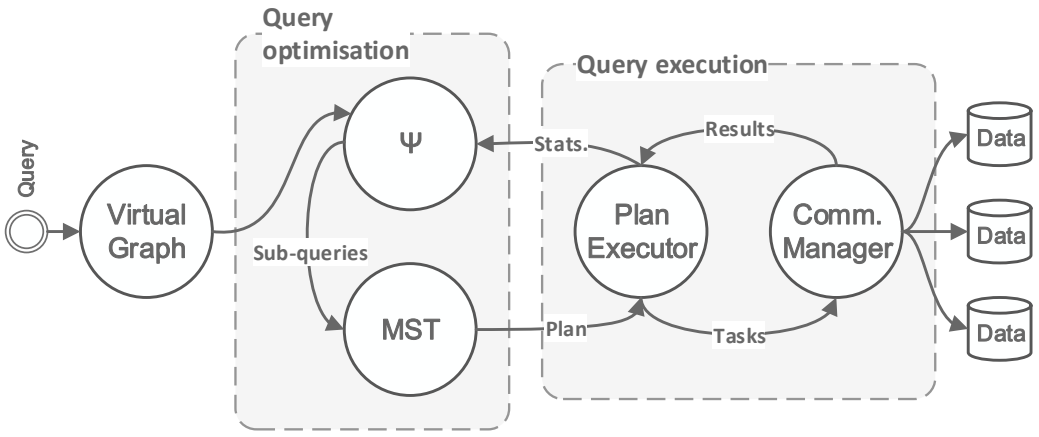


FIGURE 7.1: A SPARQL query and its co-reference are firstly transformed by Virtual Graph into a query having pre-existing bindings. The transformed query is gradually processed through optimisation-execution (OE) cycles. In each OE cycle, Ψ firstly breaks the query into sub-queries, each of which is optimised in parallel by a MST-based algorithm using runtime statistics. The runtime statistics are provided by the plan executor, after executing the QEP of the last OE cycle.

we actually have to execute both {?x foaf:knows p_0} and {?x foaf:knows p_1}. As we discussed before, this straightforward way is not practical to handle complex queries. Here we propose a model called Virtual Graph that transforms queries having co-reference into normal queries that can be executed by any distributed SPARQL engine.

Virtual Graph utilises the idea that each node, concrete or not, can be regarded as a variable with different number of values. A concrete node is regarded as a variable bound to a single value. When taking co-reference into account, a concrete vertex is regarded as a variable whose values are the union of its original URIs and all co-referent ones. In addition, triple patterns sharing the same object and subject can be regarded as a variable predicate with multiple values. These variables are called virtual nodes or virtual edges. A graph containing virtual nodes or edges is called a Virtual Graph. As shown in figure 7.2, there are two predicates between o_2 and ?x, which are combined into one virtual edge having two values. The cost of a virtual edge is calculated in the following steps. Firstly all parallel edges contained in the virtual edge are estimated using equation 5.5 and the minimum one is selected. Then remaining parallel edges are calculated again as if the previous selected triple pattern has been evaluated. The sum of cost of all parallel edges is used as the cost of the virtual edge.

The transformation of Virtual Graph is applied before query execution. Firstly, for each

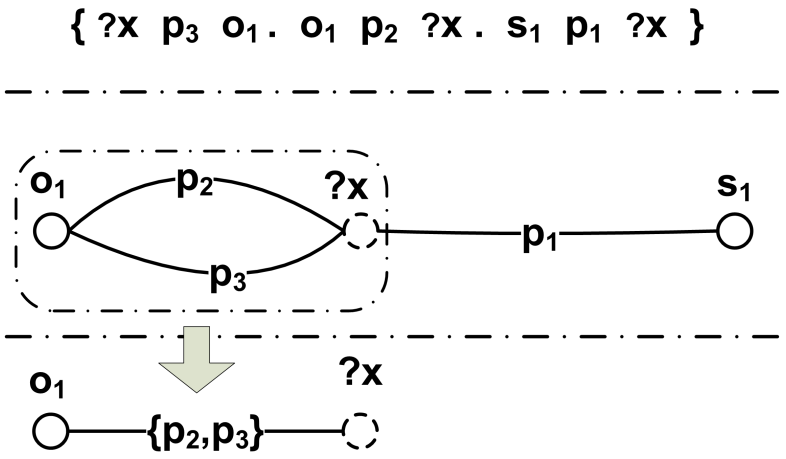


FIGURE 7.2: In the middle is the graph representation of the query on top. Solid cycles represent concrete values while dashed cycles represent variables. Bottom shows the corresponding Virtual Graph. The virtual edge has two values: p_2 and p_3.

concrete value v in a given query, our engine generates a query {v owl:sameAs ?$coref$} to all data services that may contain equivalent URIs of v. Then a variable vertex ?v is created to replace the original concrete vertex. All equivalent URIs of v, including v itself, are added to the new variable vertex ?v. The whole transformation is analogous to the process shown in figure 7.2. After this transformation, co-reference in a query is processed together in a single query, and thus generates much less query requests than using the baseline approach. In addition, it increases the possibility of benefiting from parallelism.

Meanwhile, Virtual Graph is also useful to process queries in which more than one properties exist between two resources[5]. Such queries form multigraphs that contain parallel edges, that is, edges that share the same end-nodes. For a graph without parallel edges, the optimal QEP corresponds to the MST of a query graph. For a multigraph, however, it could be a challenge to construct the optimal QEP using MST-based algorithm. From a graph theory perspective, the MST of a multigraph can be computed in two steps: 1) for parallel edges sharing the same nodes, retain only the minimum edge, and then 2) apply an ordinary MST algorithm to the modified graph. However, this approach may lead to false QEPs for SPARQL queries. The issue arises from selecting the minimum edge, that is, the triple pattern with lowest cost, out of parallel edges. Once the minimum edge is selected and executed, edges sharing nodes

[5]An example is given by the 3rd and the 5th triple patterns in Q2, which are %ProductXYZ% bsbm:producer ?p and %ProductXYZ% dc:publisher ?p respectively.

with this edge become bound and are used to prune existing bindings. Thus, the selection of the minimum edge involves not only its own cost, but also the cost of parallel edges. This behaviour is better captured by Virtual Graph. Virtual Graph is also useful for engines that adopt exhaustive search algorithms, as it reduces parallel edges into one edge.

7.4 Interleaved Query Optimisation and Execution

The presence of co-reference changes the statistics of data sources. These changes are nondeterministic for query engines since no statistics are currently available for co-referent URIs. To compensate, we propose an approach that interleaves query optimisation and execution (i.e. dynamic optimisation), to take advantage of statistics available during query processing.

In LHD-d a query is also divided into independent sub-queries using $\Psi(V,E)$ (algorithm 1), and even further. In LHD-s, only concrete nodes are regarded as fix-cardinality. In LHD-d, since the cardinality of executed triple patterns is precisely known, we are able to identify more fix-cardinality nodes using the following heuristics: 1) if the estimations of the cardinality of a variable $?v$ w.r.t all its connected triple patterns are close (i.e. for each triple pattern T_i having v, $card(T_i, ?v)$ is close to the same number) the number of bindings of $?v$ probably will not change (equation 7.1); if the number of existing bindings of $?v$ is very small, it probably will not change (equation 7.2).

$$\forall\ T_i, T_j \in conn(?v): \quad 90\% < \frac{card(T_i, ?v)}{card(T_j, ?v)} < 110\% \tag{7.1}$$

or

$$|?v| < \min_{T \in conn(?v)} (card(T, ?v))/10 \tag{7.2}$$

where $conn(?v)$ gives all triple patterns that are connected to $?v$ and $|?v|$ is the number of existing bindings of v. As in LHD-s, each sub-query is processed in parallel.

The effectiveness of the above two heuristics depends on the accuracy of cardinality estimation. In LHD-s, the estimated cardinality can be inaccurate if it is based on another estimation. Therefore, the heuristics are not used in LHD-s by default. In the

mean time, they are used in LHD-d in where actual number of bindings are used in cardinality estimation. More details will be given in chapter 7.

As a result of interleaving query optimisation and execution, the optimisation algorithm is performed more than once during query processing. With this in mind, we use a greedy algorithm in the query optimisation of LHD-d. First, greedy algorithms generally have time complexity lower than dynamic programming (or other exhaustive algorithms). Since optimisation is performed at each time an actual result size becomes available, greedy algorithms can reduce the optimisation time. Second, the accuracy advantage of dynamic programming does not hold in the circumstances of dynamic optimisation. Dynamic programming requires estimated cardinality of all triple patterns. During the construction of a QEP, it is likely the case that the cardinality of some triple patterns is estimated using both pre-computed statistics as well as the actual size of intermediate results. We have shown in chapter 5 that estimated result sizes deviate from the actual sizes. Therefore, dynamic programming is likely to change its decision (a partial QEP) when a new accurate result size becomes available. On the contrary, greedy algorithms build the optimal plan incrementally and require only actual sizes of results of previously executed triple patterns. Thus greedy algorithms can better benefit from runtime statistics.

Given a (sub-)query graph, we use the MST algorithm, shown in algorithm 4, to find the order of triple pattern execution in real time. Each time the algorithm is called, it maintains a list of remaining edges ordered by their estimated cardinality[6] from low to high. If an edge has two possible costs (i.e. if it can be executed by either a hash join and a bind join), the smaller one is chosen. Then the algorithm returns and removes the minimum edge (it belongs to the MST), which is going to be executed. It also returns edges whose subjects and objects are all bound (i.e. edges that do not belong to the MST), which are used to prune existing bindings.

The overview of query execution of LHD-d is shown as algorithm 5. Firstly a given query is broken into sub-graphs. For each sub-graph a new thread is created. At each step, minimum-cost triple patterns are selected (lines 6) and executed (line 7 to 8). Then cost of remaining edges (executed edges are removed at the end of algorithm $NextEdges(V, E)$) are updated using runtime statistics and $Execute(V, E)$ is called

[6]Responding time estimation of edges as it only needed to determine usage of parallelisation. Since $\Psi(V,E)$ takes over parallelism decision in LHD-d, we only estimate the result sizes of edges.

Algorithm 4: NextEdges(V,E)

input : A connected (sub-)graph (V, E)
output: *next* a set of edges to be executed

```
1 edges ← sort(E);
2 next ← edges[0];
3 next ← next ∪ findBoundEdges(edges);
4 E ← edges − next;
```

recursively. It should be noted that a sub-graph can be further divided in future call of $Execute(V, E)$ w.r.t updated edge cost.

Algorithm 5: Execute(V,E)

input : A connected (sub-)graph (V, E)

```
1  if E is empty then
2      return;
3  end
4  components ← Ψ(V, E);
5  foreach sub-graph (V′, E′) ∈ components create a new thread do
6      next ← NextEdges(V′, E′);
7      evaluate next[0];
8      use remaining edges of next to prune bindings;
9      update costs of edges in E′;
10     Execute(V′, E′);
11 end
```

7.5 Summary of LHD-d

In this chapter we described an engine named LHD-d that aims to provide an efficient solution for querying LD having co-reference. In LHD-d, we proposed a model called Virtual Graph for co-reference integration, and a dynamic optimisation approach to exploit runtime statistics. Virtual Graph regards a node having co-reference as a variable with pre-existing values, and transforms a query having co-reference to a regular query with pre-existing bindings. Following the transformation the query is broken into independent sub-queries that are processed in parallel. Each sub-query is processed in a recursive manner that each step of the recursion consists of an optimisation phase and an execution phase. The minimum triple pattern is identified by a MST algorithm in the optimisation phase, and is executed. Consequently, the number of results is used to re-calculate the cost of remaining triples.

7.6 Implementation

The implementation of Virtual Graph and the interleaved optimisation-execution procedure is trivial given the aforementioned descriptions. Besides, a large proportion of the infrastructure implementation of LHD-s, such as the communication manager and the Hash Bind Join operator, is reused in LHD-d. However, LHD-d uses normal hash join instead of the Double-pipelined Hash Join. This is because LHD-d requires the size of the entire results of a triple pattern before executing another one.

Chapter 8

Evaluating LHD-d

In this chapter we evaluate the query processing efficiency of LHD-d in two situations: 1) with a subset of statistics that VoID can provide[1]; and 2) having co-reference[2]. In the former situation LHD-d is compared with LHD-s and FedX, to examine the effectiveness of its dynamic optimisation approach. In the latter situation LHD-d is evaluated using the evaluation framework with added co-reference statements, and compared with the baseline approach of processing co-referenced queries. This part focuses on examining the effectiveness of Virtual Graph.

8.1 Evaluating the Dynamic Optimisation Approach

The dynamic optimisation approach enables LHD-d to take advantage of runtime statistics and thus improves the accuracy of cardinality estimation. In the meantime, it limits the ability of producing universally optimal QEPs. In this section we evaluate LHD-d using the evaluation framework and compare its results with those of LHD-s and FedX. This experiment uses the same settings as in chapter 5 (i.e. 70 million triples distributed among 20 endpoints, and detailed VoID descriptions of all endpoints). It should be noticed that LHD-d only requires the number of triples of each predicate.

[1] It does not really matter what statistics are available, since they are only used for determine the first choice in dynamic optimisation. However, in practice the most common statistics contained in VoID files are selectivity of predicates.

[2] In the evaluation having co-reference it subsumes the situation of less statistics. The intention here is to emphasise two different aspect of LHD-d.

8.1.1 Results and Analysis

We present the QPS, the incoming and outgoing traffic, and the transmission rate of engines under testing respectively in figure 8.1, 8.2, 8.3 and 8.4. "0" and "NA" stand for failures of execution. We do not include the system resource consumption of LHD-d because it is close to LHD-s. The results of LHD-s and FedX are same as those in chapter 5 since the experiment settings did not change.

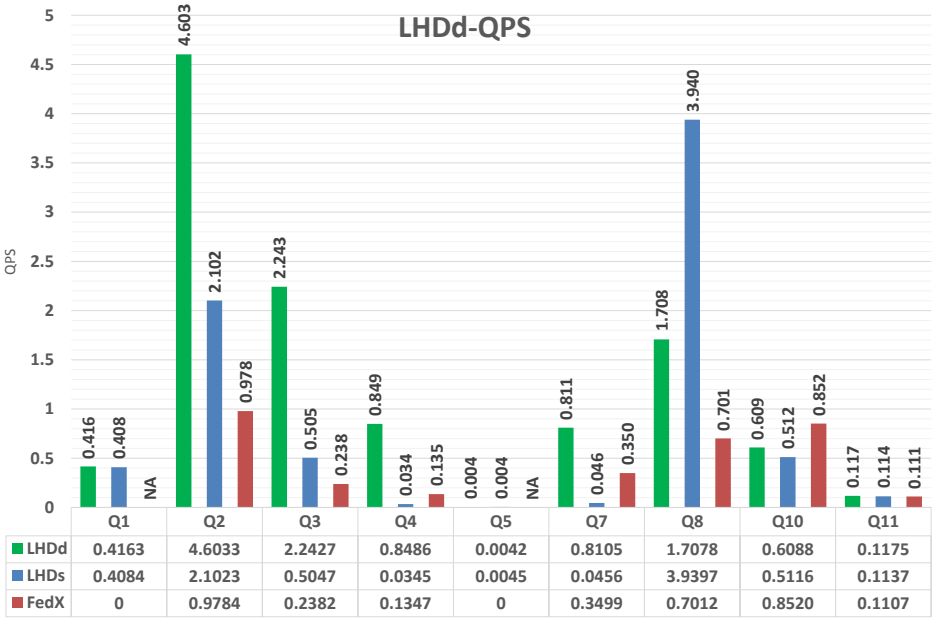


FIGURE 8.1: QPS of LHD-d

It is shown in figure 8.1 that LHD-d has an higher QPS over LHD-s on most queries (and becomes the fastest engine on most queries). Especially, significant performance boost is shown on Q2, Q3, Q4 and Q7. The boost on Q2 and Q4 is primarily due to increased transmission rate (figure 8.4), on Q3 is due to decreased network traffic (figure 8.2 and 8.3), and on Q7 is due to both factors. LHD-d is slower than LHD-s on Q8 (but still two times faster than FedX), which is due to its relatively slow transmission rate. On Q10 LHD-d shows slight improvement, but FedX is still the one with highest QPS.

LHD-d has the least network traffic on most queries except Q1, Q4 and Q10 (figure 8.2 and 8.3). It is worth noticing that in LHD-d parallelisation is determined by the Ψ algorithm (algorithm 1) in a way that network traffic is not increased. Each sub-query

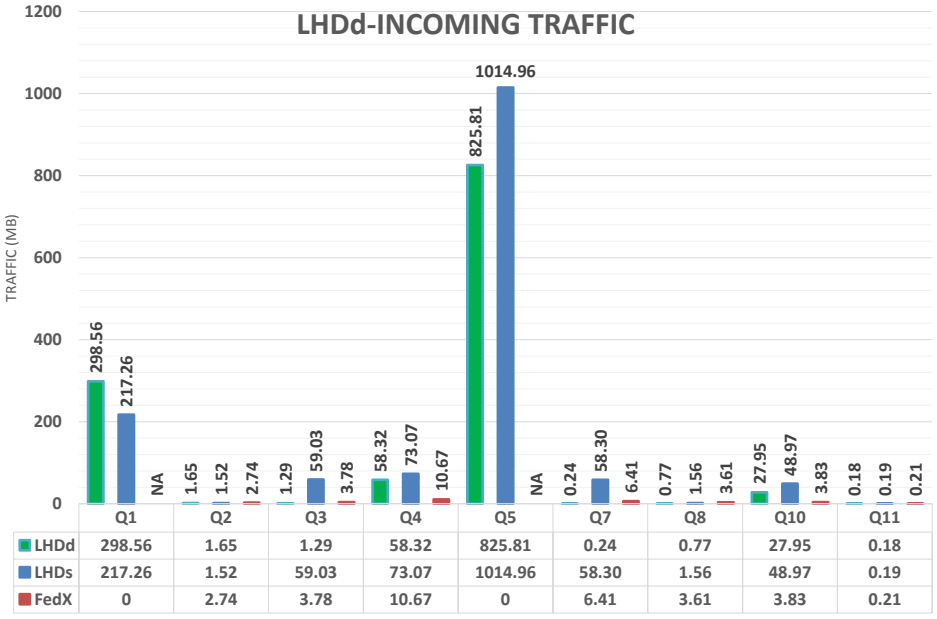


FIGURE 8.2: Incoming traffic of LHD-d

is optimised with an aim of minimum traffic. Compared with the network traffic of FedX and SPLENDID (recalling that SPLENDID produces more traffic than FedX), we conclude that using runtime statistics yields more accurate cardinality estimation and leads to QEPs that are closer to optimal. The results further reinforce the previous discussion that the existing cost models or VoID statistics are not sufficiently accurate.

The transmission rate of LHD-d varies on different queries. On Q1, Q2 and Q4 LHD-d has even higher transmission rate than LHD-s, while on Q3, Q7 and Q8 its transmission rate is relatively low. A closer look reveals that LHD-d produces insignificant amount of network traffic on Q3, Q7 and Q8, and still has highest QPS on these queries. Since LHD-d and LHD-s use the same communication management system, they have close inter-operator parallelism on simple queries, which is confirmed by the transmission rate on Q11.

In summary, the dynamic optimisation approach employed in LHD-d (i.e. using runtime statistics with the Ψ algorithm and the MST-based optimisation algorithm) better balances between reducing network traffic and increasing average transmission rate, and thus shows a higher overall efficiency. The primary advantage of LHD-d results from

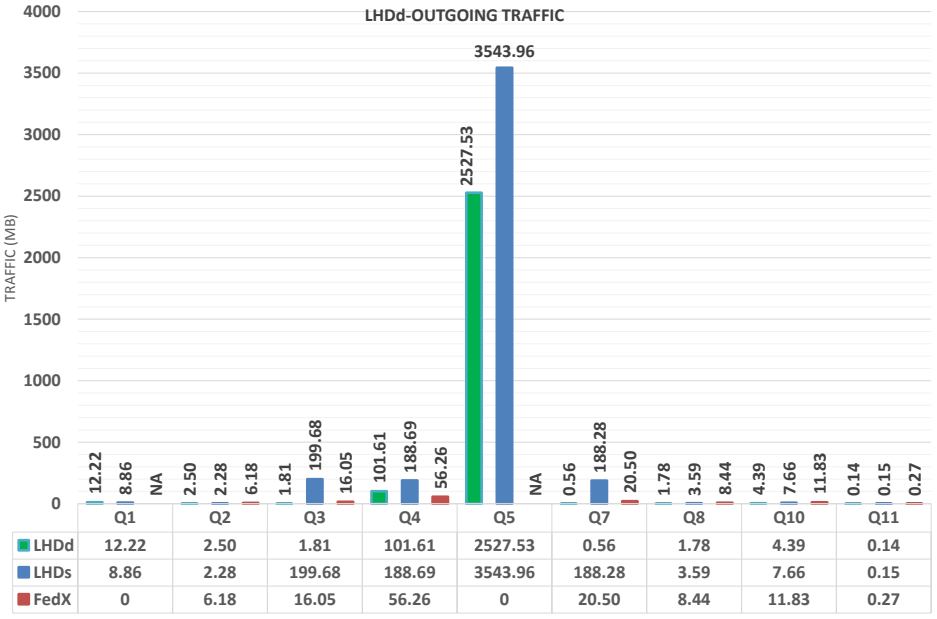


FIGURE 8.3: Outgoing traffic of LHD-d

the Ψ algorithm and the usage of runtime statistics. Runtime statistics lead to more accurate cardinality estimation than VoID statistics, and better QEPs can be produced. This consequently demonstrates that dynamic optimisation is promising for large scale LD queries, in which cases detailed statistics are difficult to obtain.

8.2 Evaluating LHD-d including Co-Reference

In this section we evaluate the efficiency of the optimisation techniques for addressing co-reference (i.e. the Virtual Graph and the aforementioned dynamic optimisation approach), that are employed in LHD-d. We compare the performance of LHD-d and the number of query results in situations that are with or without co-reference, to explore the impact of co-reference. In addition, we compare LHD-d with the baseline approach of processing co-referenced queries, to demonstrate the effectiveness of Virtual Graph. The above-mentioned two evaluations has demonstrated that: 1) Taking co-reference into distributed SPARQL queries yields a large amount of supplementary results, but also significantly increase query responding time; and 2) the concept of Virtual Graph

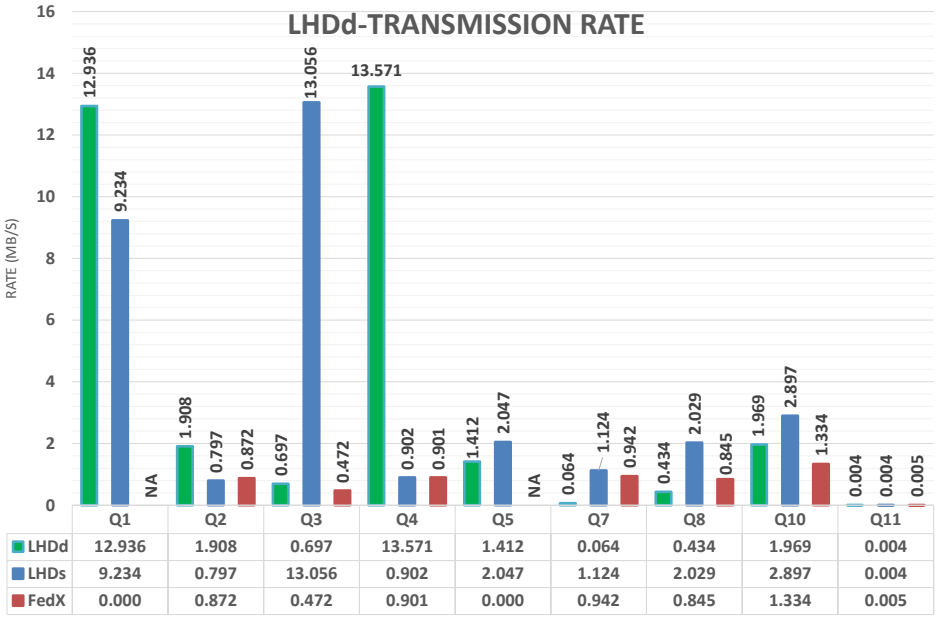


FIGURE 8.4: Average transmission rate of LHD-d

for BGPs (together with the dynamic optimisation approach) effectively reduce the time required for processing co-reference of queries.

8.2.1 Experiment Settings

This experiment continues to use the same settings as the previous one (i.e. 70 million triples distributed among 20 SPARQL endpoints) with 0.18 million additional co-reference statements (0.25% of 70 million triples). These co-reference statements are generated following the method described in section 4.2, based on the proportion and distribution of co-reference in the real world. In addition, all *LIMIT* modifiers are removed from the query set to show the extra results brought by co-reference.

The baseline approach with which LHD-d is compared first generates the Cartesian product of all co-referent URIs (including the original URIs) in a given query. For each entry of the Cartesian product a new query is created by replacing the concrete URIs in the query having co-reference of the entry. Finally every new query is executed using LHD-d without Virtual Graph turned on. The union of the results of all co-referent

queries is regarded as the result of the baseline approach. In other words, the baseline approach only differs from LHD-d on the usage of Virtual Graph.

8.2.2 Results and Analysis

In the remainder of this section we use LHD-d* to represent the evaluation results of LHD-d obtained with the presence of co-reference, and LHD-d to represent the results without co-reference taken into account.

We show in table 8.1 that both LHD-d and the baseline approach produce the same sizes of results having co-reference taken into account. This confirms the ability of Virtual Graph to fully retrieve additional results due to co-reference. Meanwhile, the result sizes are raised many times (even orders of magnitude on specific queries) by the small proportion of additional co-reference statements. The result sizes of Q5 and Q11 remain the same for different reasons. Q5 selects for products that share the same feature with a given product. There are 14499 distinct products in our dataset, all of which are already contained in the result of Q5 without co-reference. By turning on co-reference support in LHD-d, many more intermediate results are generated (demonstrated by the network traffic of Q5 in figure 8.6), but the final result does not change. The reason for Q11 is straightforward. Q11 does not have concrete subjects or objects, so its result size remains the same.

Three reasons are relevant to the significant amount of additional results. First, a single vocabulary is shared by all endpoints. Second, in our datasets co-reference exists between instances of all classes (e.g. Products, Product Features). Consequently, Cartesian product of a large size is probably produced by the co-reference of the concrete subjects and objects in a query. Third, instances of the same class have similar relationships with instances of other classes. Therefore, each co-referent URI may well lead to a valid result.

In the real world, domains, in where datasets have a similar structure as the dataset in our experiment, are likely to gain the same boost of results by supporting co-reference in distributed SPARQL engines. In domains having only part of the above three conditions,

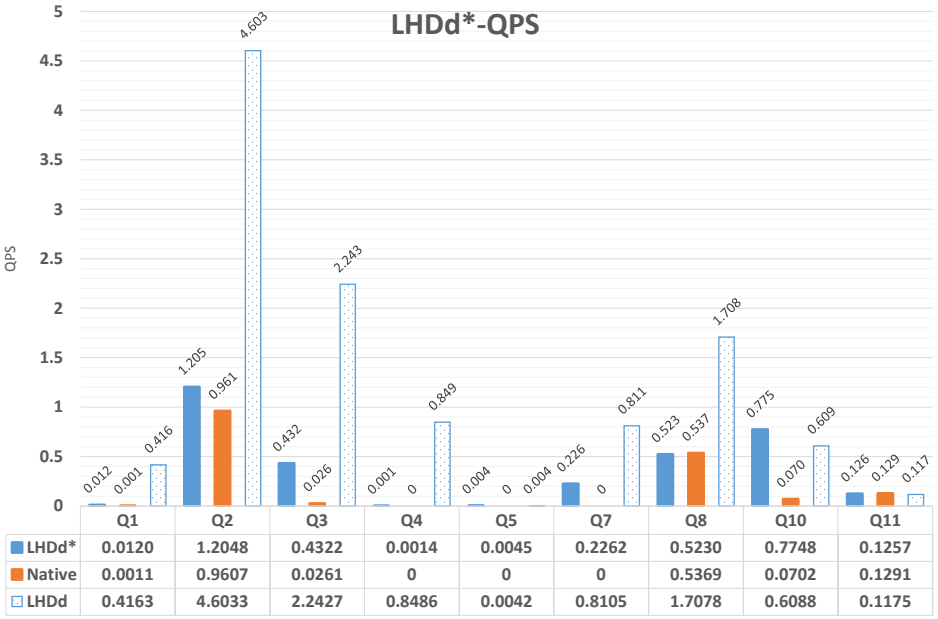
it is unknown whether the same amount of extra result will be produced by taking co-reference into account. Investigating the structure of datasets which are connected by co-reference of different domains is in our future plan.

TABLE 8.1: Comparison of result sizes with or without co-reference. The first columns represent result sizes with the presence of co-reference returned by LHD-d* and the naive approach while the last column represents result sizes without the presence of co-reference returned by LHD-d. It is clear that co-reference significantly increase result sizes.

Query	**LHD-d***	**Naive**	**LHD-d**
Q1	7397	7397	53
Q2	103	103	29
Q3	23	23	8
Q4	65510	NA	29
Q5	14499	NA	14499
Q7	1579	NA	63
Q8	101	101	21
Q10	32	32	12
Q11	10	10	10

We present the QPS, the incoming and outgoing traffic, and the transmission rate of LHD-d*, LHD-d, and the baseline approach respectively in figure 8.5, 8.6, 8.7 and 8.8. "0" and "NA" stand for failures of execution.

It is shown in figure 8.5 that the efficiency of query processing is decreased multifold after introducing co-reference. Especially, the QPS of the baseline approach is orders of magnitude lower than that of LHD-d (without co-reference). Moreover, the baseline approach fails on several queries (Q4, Q5 and Q7) that have a large result size. Although LHD-d* still has low QPS on a few queries, it substantially increases the efficiency of co-reference query processing.On Q10 LHD-d* has an even higher QPS than LHD-d, indicating a good QEP that overcomes the negative effect of co-reference, is generated. Q11 has no co-reference, and the three approaches show close QPS. This indicates that the impact on QPS solely comes from introducing co-reference rather than the modification of query engines. Recalling that the usage of Virtual Graph is the only difference between LHD-d* and the baseline approach, we conclude that processing all



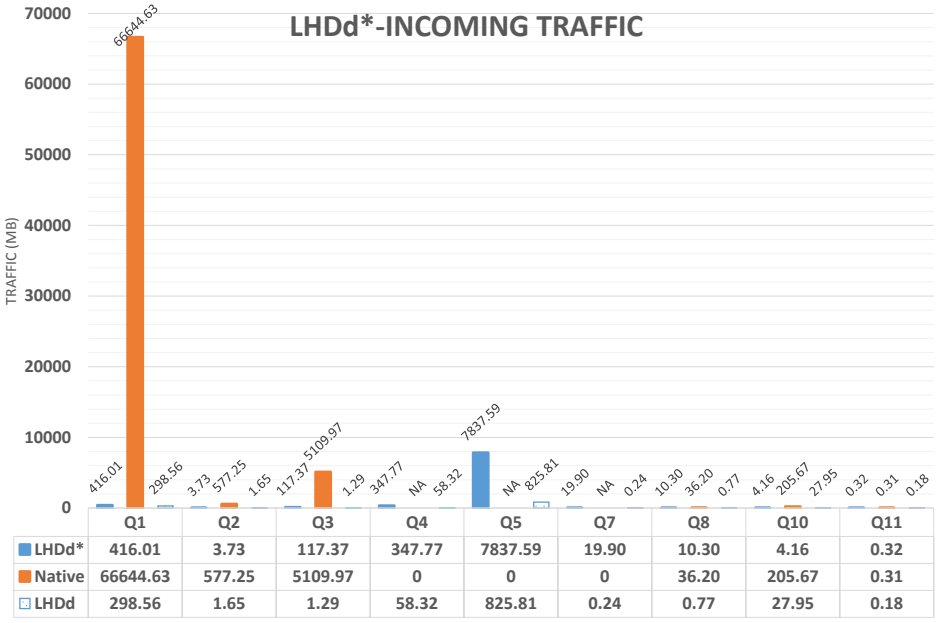
	Q1	Q2	Q3	Q4	Q5	Q7	Q8	Q10	Q11
LHDd*	0.0120	1.2048	0.4322	0.0014	0.0045	0.2262	0.5230	0.7748	0.1257
Native	0.0011	0.9607	0.0261	0	0	0	0.5369	0.0702	0.1291
LHDd	0.4163	4.6033	2.2427	0.8486	0.0042	0.8105	1.7078	0.6088	0.1175

FIGURE 8.5: QPS of LHD-d having co-reference (LHD-d*)

co-referent URIs together improves query efficiency. Later we will provide evidence that this improvement is because the Virtual Graph enables LHD-d* not only to process more co-referent queries at a time, but also to produce the optimal QEPs w.r.t all co-referent URIs. On the contrary, although the baseline approach produces optimal plans for each co-referent query, the total query time is not necessarily minimised.

From the network traffic of both LHD-d* and the baseline approach (figure 8.6 and 8.7) it is shown that co-reference increase the sizes of intermediate results to a large extent. In the meantime, LHD-d* shows much less amount of network traffic compared to the baseline approach. It confirms that Virtual Graph enables LHD-d* to find better QEPs that lead to lower network traffic. Otherwise, if the same QEPs were followed by both LHD-d* and the baseline approach, higher transmission rate may occur since the Virtual Graph enables more co-referent queries to be processed at the same time. However, the amount of network traffic would not change much.

LHD-d* and the baseline approach have the same amount of traffic on Q11, which is slightly larger than that of LHD-d. Along with the same transmission rate of LHD-d* and the baseline approach on Q11 (figure 8.8), it can be confirmed that both cases

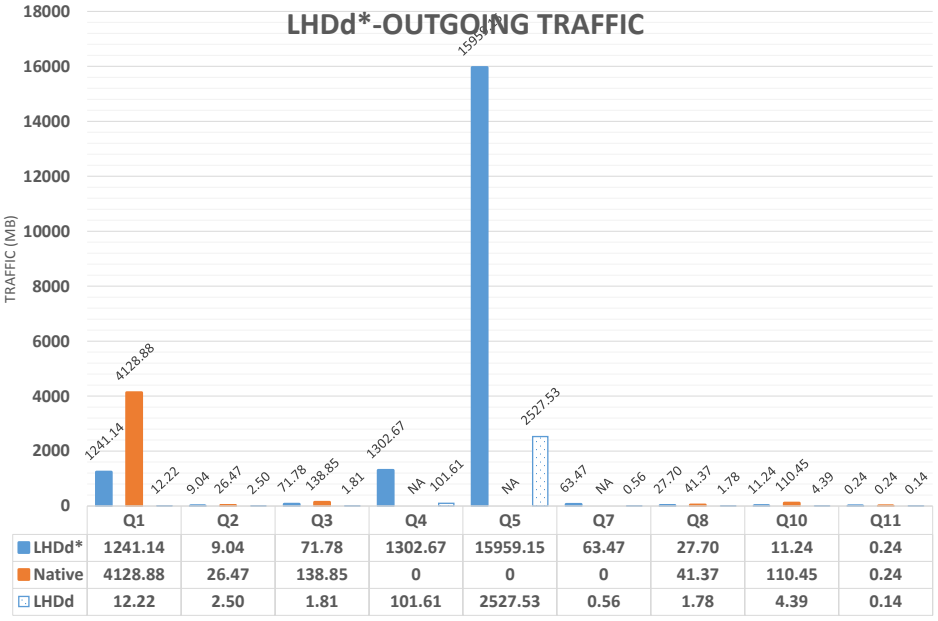


	Q1	Q2	Q3	Q4	Q5	Q7	Q8	Q10	Q11
LHDd*	416.01	3.73	117.37	347.77	7837.59	19.90	10.30	4.16	0.32
Native	66644.63	577.25	5109.97	0	0	0	36.20	205.67	0.31
LHDd	298.56	1.65	1.29	58.32	825.81	0.24	0.77	27.95	0.18

FIGURE 8.6: Incoming traffic of LHD-d having co-reference (LHD-d*)

have the same behaviour. The extra traffic and transmission rate over LHD-d is due to searching for co-reference of Q11 (although no co-referent URIs are found).

The transmission rate of LHD-d* is not always higher than that of LHD-d and the baseline approach on different queries. This further confirms that Virtual Graph enables LHD-d* to generate QEPs especially tailored w.r.t all co-referent queries. It is because at any step during query execution more traffic is generated by LHD-d* since all co-referent URIs are processed together. If the same QEPs were generated in LHD-d*, the transmission rate of LHD-d* would always be no less than LHD-d and the baseline approach.



	Q1	Q2	Q3	Q4	Q5	Q7	Q8	Q10	Q11
LHDd*	1241.14	9.04	71.78	1302.67	15959.15	63.47	27.70	11.24	0.24
Native	4128.88	26.47	138.85	0	0	0	41.37	110.45	0.24
LHDd	12.22	2.50	1.81	101.61	2527.53	0.56	1.78	4.39	0.14

FIGURE 8.7: Outgoing traffic of LHD-d having co-reference (LHD-d*)

8.3 Evaluation Summary

In this chapter we evaluate a combination of optimisation techniques for environments with co-reference. These techniques include a Virtual Graph model to address co-reference, a MST-based algorithm to dynamically optimises queries using runtime statistics, and the Ψ algorithm that potentially identifies more parallel sub-queries using runtime statistics. The evaluation clearly demonstrates the effectiveness of Virtual Graph on improving query performance having co-reference taken into account. The performance gain is due to two factors: 1) Virtual Graph reduces the number of requests required for addressing co-referent URIs; 2) Virtual Graph enables the optimiser to find the optimal QEP w.r.t all co-referent queries of a given query. In the meantime, the evaluation further demonstrates the advantage of dynamic optimisation in environments where accurate statistics are not available.

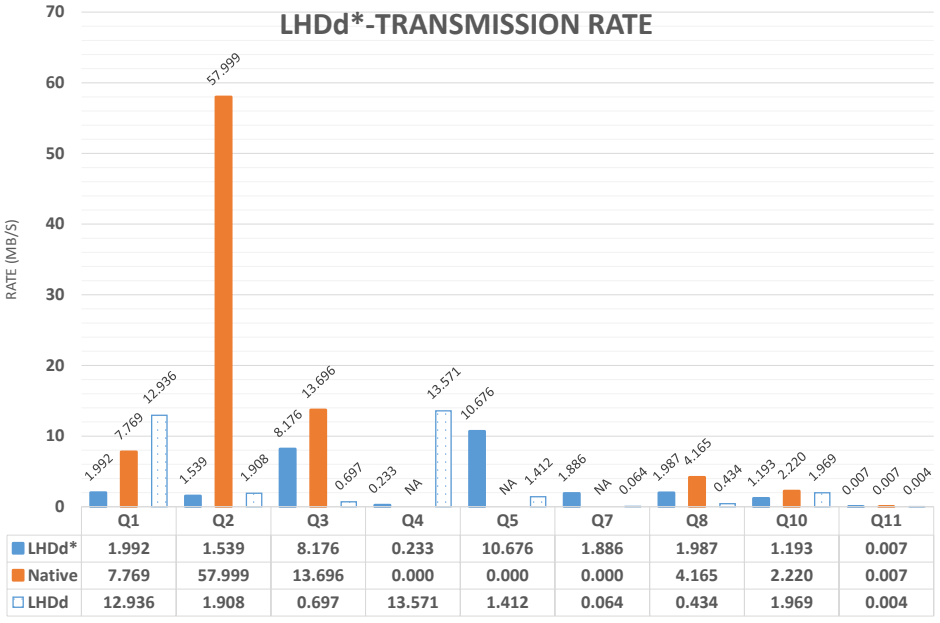


FIGURE 8.8: Average transmission rate of LHD-d having co-reference (LHD-d*)

Chapter 9

Conclusions and Future Work

The uprising Web of Data leads to a new era of data consuming, in which web applications are able to understand information and cooperate with humans on complex tasks. Modern web applications have already shown the trend of rich data interaction, which relies on low latency queries. The Semantic Web technologies potentially raise the interaction between applications and data to a Web scale. Tasks involving federation of multiple datasets, such as the example described in chapter 1, will no longer limited by individual or organisation boundaries. Lucy will be able to find the most appropriate doctor for her mother on this planet (if by that time remote medical treatment will no more be a problem). The loss of data boundaries will in turn stimulate more sophisticated applications that are capable for tasks more complex than looking for doctors. There is a strong demand for approaches that can efficiently query the Web of Data.

Querying the Web of Data, or distributed SPARQL query processing, can benefit from the developments in distributed DBMS. However, most distributed DBMS derive their efficiency from reliable connections among datasets, predictable data structure and controlled statistics, neither of which can be expected from LD due to its large scale and distributed nature. Besides, co-reference, the phenomenon of the same resource referred to by multiple URIs, puts forward unique challenges to querying LD.

Motivated by the above demands, we investigated schemes that jointly use novel techniques that are tailored for distributed SPARQL queries, as well as distributed DBMS

techniques. We propose two sets of optimisation techniques, implemented in two distributed SPARQL engines, LHD-s and LHD-d, for typical scenarios on the Web of Data.

In addition, we propose DSEF, a scalable and flexible evaluation framework for distributed SPARQL queries. Using DSEF we compared the LHD schemes with other approved approaches. Based on the results we established the open issues of existing query processing techniques and propose promising alternatives.

In the following we firstly summarise DSEF and the two LHD schemes, followed by conclusions regarding distributed SPARQL query processing. In addition we describe our future plans based on the open issues revealed by our evaluation.

9.1 Summary of DSEF

DSEF is a benchmark tailored for evaluating distributed SPARQL engines in networks of arbitrary scales. The flexibility and scalability of DSEF derive from a VM-based network architecture and the use of artificial data. Moreover, DSEF uniquely introduces the ability of simulating co-reference in a given RDF network, based on real-world proportion and distribution of co-reference.

The aforementioned features are backed by a set of scalable and efficient tools, which provide convenient functionalities including:

- Generating RDF data of arbitrary sizes, with optional co-reference statements (i.e. *owl:sameAs* triples).
- Dividing RDF data into smaller pieces according to a given distribution.
- Producing detailed VoID files for given RDF data.
- Simultaneously uploading data to remote datasets, with the ability to resume interrupted transmission.
- Automatically testing given engines and generating reports as “csv” files.

These tools enable quick setting up of required experimental environments, and gathering statistics of tested engines.

DSEF adopts widely accepted BSBM data and queries to establish soundness. The assessment queries are carefully adjusted in a way that better explores inner mechanism of tested engines in distributed settings, while retaining the queries' original semantics. DSEF includes three primary metrics:

- Query per second (QPS), represents the average number of queries executed per second.
- Network traffic, represents the total amount of network traffic (both incoming and outgoing) produced due to executing queries.
- Transmission rate, represents the average speed of network communication. It is calculated as the network traffic divided by the query execution time.

In particular DSEF further includes two secondary metrics to monitor system resource consumptions:

- CPU usage, presents the average percentage of CPU used to execute a certain query.
- Memory usages, presents the average amount of memory used to execute a certain query.

9.2 Summary of LHD-s

LHD-s is a distributed SPARQL engine developed based on a scheme of techniques that are tailored for RDF networks with detailed VoID statistics.

The VoID statistics contain the number of triples, distinct subjects and objects per predicate, and are used by a selectivity-based responding time cost model. In the cost model we use a new method to estimate cardinality of joined triple patterns, which is the basis of cost estimation. We demonstrate that on queries that have two triple patterns and no concrete subjects or objects, the proposed method outperforms existing approaches on the DSEF environment. The effectiveness of our cost model over more complex queries is unknown, due to the significant complexity of performing experiments with all possible queries.

LHD-s follows a static optimisation approach. A dynamic-programming-based optimisation algorithm is used to select the QEP having the minimum responding time. While dynamic programming guarantees to find the optimal plans per cost model, it has a high growth of order of complexity. To further improve its efficiency, we take advantage of certain join operators and introduce heuristics to reduce complexity without decreasing QEP quality (details are mentioned in section 5.6.1). Furthermore, the optimisation of LHD-s is able to produce QEPs for parallel execution using cost models that are not parallelism aware. This is achieved by transforming the QEP, generated in the aforementioned step, to its parallel form. This parallel transformation only reduces the executing time of the original QEP, without increasing other costs unchanged.

Parallelism is intensively used through both query optimisation and execution of LHD-s. We introduce the side-effect-free parallelism, which increases the degree of parallelism without increasing network traffic. More specifically, we propose an efficient (in terms of time complexity) algorithm called Ψ (parallel sub-query identification, algorithm 1), to identify sub-queries that can be optimised and executed independently from each other, by analysing the invariance of cardinality of variables (i.e. fixed-cardinality nodes). It is worth mentioning that the Ψ algorithm is applied to queries, while the parallel transformation in optimisation is applied to QEPs. In terms of occurrence time, the former takes place at the beginning of the whole query processing, while the latter happens after a QEP is produced. In LHD-s, parallelism is resolved by both Ψ and parallel QEPs.

LHD-s provides a parallel query execution system that is able to maximise the transmission rate for given QEPs. It decouples the logical execution of QEPs and physical communication with RDF datasets. The former is regulated by a plan executor that starts execution of a triple pattern as soon as its depending bindings are ready. The execution does not directly contact remote endpoints, but submits execution tasks to a communication manager, which controls physical communications with RDF datasets. The number of concurrent connections to each dataset is individually maintained, w.r.t the available bandwidth to and the computing power of the dataset. Thus LHD-s is able to exploit transmission rate to the uttermost. Once a triple pattern is executed, its bindings are delivered back to the plan executor and pushes forward execution of remaining triple patterns.

9.3 Summary of LHD-d

In contrast to LHD-s, LHD-d is developed based on a scheme of techniques that are designed for RDF networks having co-reference, and less accurate statistics.

Co-reference is taken into account in LHD-d using a model called Virtual Graph. Virtual Graph considers a concrete URI having co-reference as a variable with pre-existing values which include the original URI and its co-reference. By applying the Virtual Graph, a query and its co-reference are evaluated collectively as a regular single query with pre-existing bindings. We provide evidence that Virtual Graph not only saves the effort of evaluating each co-reference query individually, but also enables query optimisation to take all co-reference into account simultaneously.

The presence of co-reference alters data statistics in a nondeterministic fashion. Instead of obtaining statistics of RDF datasets from VoID files, LHD-d primarily relies on statistics that become available at runtime. After a triple pattern is executed, its accurate result size is known, which is used to estimate costs of remaining triple patterns. This method prevents propagation of estimation errors of early stages, and minimises the possibility of bad choices.

Due to the usage of runtime statistics, LHD-d follows a dynamic optimisation approach, in which query optimisation and execution are interleaved. Since the result sizes keep updating in each optimisation-execution cycle (OE cycle), LHD-d adopts a MST-based algorithm to select the minimum remaining triple pattern (in terms of estimated result size). The MST-based optimisation focuses on reducing network traffic (since it is difficult to resolve parallelism within an OE cycle), and leaves decisions of parallelism to Ψ.

The Ψ algorithm is exploited even further in LHD-d. First, Ψ is applied at the beginning of each OE cycle, and fully determines the parallelism of LHD-d. Second, benefiting from runtime statistics, two heuristics for determining fixed-cardinality variables are added. As more fixed-cardinality variables are likely to be found, the chance of having a higher degree of parallelism is increased.

LHD-d adopts the same parallel execution system as in LHD-s. The only difference is that LHD-d uses normal Hash Joins in the places where Double-pipelined Hash Joins

(DHJs) are used. This is because LHD-d requires the size of all results of a triple pattern, which is not available from DHJs.

9.4 Conclusions

The main conclusion following the evaluation results is that either or both VoID statistics and existing selectivity-based cost models are not sufficiently accurate. It is drawn from two observations: 1) LHD-s optimises queries for minimum responding time, however, it is slower than LHD-d; and 2) SPLENDID, whose optimisation objective is minimum network traffic, produces more traffic than FedX, whose optimisation is based on heuristics. Recalling that both LHD-s and SPLENDID adopt dynamic programming, VoID statistics and cost models are the only remaining sources leading to sub-optimal plans.

In this thesis we tested our approaches in an environment having up to 20 endpoints. In the LD cloud this number could be larger and it would be more difficult to maintain accurate statistics. Based on the previous conclusion and the encouraging results of LHD-d, we in addition conclude that using runtime statistics and dynamic optimisation is promising for LD queries in general. Furthermore, the effectiveness of Ψ algorithm is better exploited with dynamic optimisation, which better balances transmission rate and traffic.

Informatively, we notice that some types of queries significantly benefit from query optimisation while others are difficult to optimise. For example, Q5 generates a large amount of intermediate results regardless of the optimisation techniques. We call such queries inefficient queries. For such queries more efficient execution techniques will play an important part to reduce query response time and optimisation will probably increase query response time. Furthermore, although it can be difficult to investigate general rules to identify inefficient queries before execution (until accurate statistics and cost models are developed), it is possible to provide guidelines to SPARQL users to avoid such queries, by studying the structure and composition of queries. A trivial example will be the query {?s ?p ?o}, which will return everything from a dataset. A more subtle case is that star shaped queries with variables at the centre are more likely to generate a large amount of intermediate result than chain shaped queries. Graph theory is likely to provide insight on this subject.

9.5 Future Work

Despite the encouraging results we described in this thesis, there is still a variety of work required for both further improving the performance of LHD-s and LHD-d, and exploring open issues of distributed SPARQL processing in general. Following these two lines, this section breaks our future plans into two segments.

9.5.1 Short-term Plans on Improving the Proposed Methods

The most urgent enhancement for LHD-s is to improve the accuracy of cost estimation. However, since it is related to statistics-based engines in general, we leave it to the section in which general issues are discussed.

In the near future, both LHD-s and LHD-d will be evaluated in the real world. Such experiment will provide more comprehensive understanding of adopted techniques and clues for further enhancements. Especially, for specific domains, it is possible to take advantage of knowledge that is not widely available, and to adopt specific optimisation. A promising direction could be extending the work described in Akar et al. [2012], to filter out groups of triple patterns that will not produce any result. Moreover, due to the limitation of our evaluation with co-reference, it is unknown in which domains a small portion of co-reference will lead to a large number of additional results. The correlation between the structure of co-reference and the effectiveness of Virtual Graph is also unknown. Real-world evaluation will be complementary to the lab based benchmarking presented in this thesis and provide more comprehensive insight of distributed SPARQL optimisation. Further studies will need to be conducted to take both evaluation into account.

In the LHD schemes we focus on techniques of query optimisation and parallelism. To become fully practical distributed engines, assistant techniques such as caching are necessary. In addition, since the aggregation of BGPs in Jena constrains the usage of parallelism, it is worth implementing the aggregation in a parallelism compatible way.

9.5.2 Long-term Plans on Open Issues of Distributed SPARQL

The main open issue exposed by this thesis is the lack of effective and concise statistics of RDF datasets, or accurate cost models. In section 6.1 we described experiments to analyse the characteristics of cardinality of complex joins. This is the first step of investigating the aforementioned open issue. There are two possible directions depending on the results. The first direction is looking for cost models that do not necessarily produce accurate estimations, but correctly compare two arbitrary joins based on existing VoID statistics. However, recalling the vast number of sophisticated indices in distributed DBMS, this direction could be impossible due to the inherent limitations of VoID. To this end, the second direction is to identify the most essential statistics required for comparing arbitrary joins, and extend VoID with the ability to include those statistics. Those statistics have to be concise to be widely available on the Web of Data.

In the meantime, as shown by the evaluation, a promising direction for future distributed SPARQL processing is to exploit runtime statistics with dynamic optimisation. Since in dynamic optimisation only the initial choice is made from pre-existing statistics (even the initial choice can be made purely using heuristics, just like FedX), the responsibility of providing detailed statistics is relieved from data providers. The primary future work on exploiting runtime statistics is to explore sampling techniques that retrieve statistics without waiting for finishing a triple pattern, which consequently enables engines to use pipelined parallelism (e.g. Double-pipelined Hash Join). Besides, to develop more sophisticated algorithms for dynamic optimisation is also an important part of our future work.

The use of pipelined parallelism not only elevates the efficiency of LD queries, but also increases adaptivity of query processing. Adaptive query processing has been well developed in distributed DBMS, however, limited work has been done for distributed SPARQL (as far as we know, ANAPSID [Acosta et al., 2011] is the only work on adaptive SPARQL evaluation). The Web of Data contains RDF datasets of various kinds, and adaptivity is no less important than efficiency.

Appendix A

Experiment Queries

Table A.1: Assessment queries of the evaluation framework

Query 1

```
SELECT DISTINCT ?product ?label
WHERE {
?product rdfs:label ?label .
?product a %ProductType% .
?product bsbm:productFeature %ProductFeature1% .
?product bsbm:productFeature %ProductFeature2% .
?product bsbm:productPropertyNumeric1 ?value1 .
}
ORDER BY ?label
LIMIT 10
```

Continued on next page

Table A.1 – continued from previous page

Query 2

```
SELECT ?label ?comment ?producer ?productFeature
?propertyTextual1 ?propertyTextual2 ?propertyTextual3
?propertyNumeric1 ?propertyNumeric2 ?propertyTextual4
?propertyTextual5 ?propertyNumeric4
WHERE {
%ProductXYZ% rdfs:label ?label .
%ProductXYZ% rdfs:comment ?comment .
%ProductXYZ% bsbm:producer ?p .
?p rdfs:label ?producer .
%ProductXYZ% dc:publisher ?p .
%ProductXYZ% bsbm:productFeature ?f .
?f rdfs:label ?productFeature .
%ProductXYZ% bsbm:productPropertyTextual1 ?propertyTextual1 .
%ProductXYZ% bsbm:productPropertyTextual2 ?propertyTextual2 .
%ProductXYZ% bsbm:productPropertyTextual3 ?propertyTextual3 .
%ProductXYZ% bsbm:productPropertyNumeric1 ?propertyNumeric1 .
%ProductXYZ% bsbm:productPropertyNumeric2 ?propertyNumeric2 .
}
```

Continued on next page

Table A.1 – continued from previous page

Query 3

```
SELECT ?product ?label
WHERE {
?product rdfs:label ?label .
?product a %ProductType% .
?product bsbm:productFeature %ProductFeature1% .
?product bsbm:productPropertyNumeric1 ?p1 .
?product bsbm:productPropertyNumeric3 ?p3 .
?product bsbm:productFeature %ProductFeature2% .
?product rdfs:label ?testVar .
}
ORDER BY ?label
LIMIT 10
```

Continued on next page

Table A.1 – continued from previous page

Query 4

```
SELECT DISTINCT ?product ?label ?propertyTextual
WHERE {
{
?product rdfs:label ?label .
?product rdf:type %ProductType% .
?product bsbm:productFeature %ProductFeature1% .
?product bsbm:productFeature %ProductFeature2% .
?product bsbm:productPropertyTextual1 ?propertyTextual .
?product bsbm:productPropertyNumeric1 ?p1 .
} UNION {
?product rdfs:label ?label .
?product rdf:type %ProductType% .
?product bsbm:productFeature %ProductFeature1% .
?product bsbm:productFeature %ProductFeature3% .
?product bsbm:productPropertyTextual1 ?propertyTextual .
?product bsbm:productPropertyNumeric2 ?p2 .
}
}
ORDER BY ?label
OFFSET 5
LIMIT 10
```

Continued on next page

Table A.1 – continued from previous page

Query 5

```
SELECT DISTINCT ?product ?productLabel
WHERE {
?product rdfs:label ?productLabel .
%ProductXYZ% bsbm:productFeature ?prodFeature .
?product bsbm:productFeature ?prodFeature .
%ProductXYZ% bsbm:productPropertyNumeric1 ?origProperty1 .
?product bsbm:productPropertyNumeric1 ?simProperty1 .
%ProductXYZ% bsbm:productPropertyNumeric2 ?origProperty2 .
?product bsbm:productPropertyNumeric2 ?simProperty2 .
}
ORDER BY ?productLabel
LIMIT 5
```

Continued on next page

Table A.1 – continued from previous page

Query 7

```
SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle
?review ?revTitle ?reviewer ?revName ?rating1 ?rating2
WHERE {
%ProductXYZ% rdfs:label ?productLabel .
?offer bsbm:product %ProductXYZ% .
?offer bsbm:price ?price .
?offer bsbm:vendor ?vendor .
?vendor rdfs:label ?vendorTitle .
?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
?offer dc:publisher ?vendor .
?offer bsbm:validTo ?date .
?review bsbm:reviewFor %ProductXYZ% .
?review rev:reviewer ?reviewer .
?reviewer foaf:name ?revName .
?review dc:title ?revTitle .
}
```

Continued on next page

Table A.1 – continued from previous page

Query 8

```
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName
?rating1 ?rating2 ?rating3 ?rating4
WHERE {
?review bsbm:reviewFor %ProductXYZ% .
?review dc:title ?title .
?review rev:text ?text .
?review bsbm:reviewDate ?reviewDate .
?review rev:reviewer ?reviewer .
?reviewer foaf:name ?reviewerName .
?review bsbm:rating1 ?rating1 .
?review bsbm:rating2 ?rating2 .
?review bsbm:rating3 ?rating3 .
?review bsbm:rating4 ?rating4 .
}
ORDER BY DESC(?reviewDate)
LIMIT 20
```

Continued on next page

Table A.1 – continued from previous page

Query 10

```
SELECT DISTINCT ?offer ?price
WHERE {
?offer bsbm:product %ProductXYZ% .
?offer bsbm:vendor ?vendor .
?offer dc:publisher ?vendor .
?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#US> .
?offer bsbm:deliveryDays ?deliveryDays .
?offer bsbm:price ?price .
?offer bsbm:validTo ?date .
}
ORDER BY xsd:double(str(?price))
LIMIT 10
```

Query 11

```
SELECT ?property ?hasValue ?isValueOf
WHERE {
{ %OfferXYZ% ?property ?hasValue }
UNION
{ ?isValueOf ?property %OfferXYZ% }
}
```

TABLE A.2: SS joins of distinct predicates

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
bsbm:productPropertyNumeric1>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
a  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyNumeric1>  ?o1 ;
a  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyNumeric1>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  a  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
bsbm:producer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
bsbm:productPropertyTextual1>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
rdfs:comment>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
bsbm:productPropertyTextual2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
bsbm:productPropertyTextual3>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:producer>  ?o1 ;
bsbm:productPropertyTextual1>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:producer>  ?o1 ;
rdfs:comment>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:producer>  ?o1 ;
bsbm:productPropertyNumeric1>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:producer>  ?o1 ;
bsbm:productPropertyTextual2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:producer>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:producer>  ?o1 ;
bsbm:productPropertyTextual3>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:producer>  ?o1 ;
rdfs:label>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:producer>  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual1>  ?o1 ;
rdfs:comment>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual1>  ?o1 ;
bsbm:productPropertyNumeric1>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual1>  ?o1 ;
bsbm:productPropertyTextual2>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual1>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual1>  ?o1 ;
bsbm:productPropertyTextual3>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual1>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual1>  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  rdfs:comment>  ?o1 ;
bsbm:productPropertyNumeric1>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  rdfs:comment>  ?o1 ;
bsbm:productPropertyTextual2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  rdfs:comment>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  rdfs:comment>  ?o1 ;
bsbm:productPropertyTextual3>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  rdfs:comment>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  rdfs:comment>  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyNumeric1>  ?o1 ;
bsbm:productPropertyTextual2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyNumeric1>  ?o1 ;
dc:publisher>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyNumeric1>  ?o1 ;
bsbm:productPropertyTextual3>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyNumeric1>  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual2>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual2>  ?o1 ;
bsbm:productPropertyTextual3>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual2>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual2>  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  dc:publisher>  ?o1 ;
bsbm:productPropertyTextual3>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  dc:publisher>  ?o1 ;
rdfs:label>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  dc:publisher>  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual3>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual3>  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  rdfs:label>  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:productFeature>  ?o1 ;
bsbm:productPropertyNumeric3>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyNumeric1>  ?o1 ;
bsbm:productPropertyNumeric3>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  a  ?o1 ;
bsbm:productPropertyNumeric3>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  rdfs:label>  ?o1 ;
bsbm:productPropertyNumeric3>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:productPropertyTextual1>  ?o1 ;
a  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  a  ?o1 ;
bsbm:productPropertyNumeric2>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
bsbm:country>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
bsbm:vendor>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
bsbm:price>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
bsbm:reviewFor>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
bsbm:validTo>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
dc:publisher>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
bsbm:product>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
dc:title>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:country>  ?o1 ;
bsbm:vendor>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:country>  ?o1 ;
bsbm:price>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:country>  ?o1 ;
bsbm:reviewFor>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:country>  ?o1 ;
bsbm:validTo>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:country>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:country>  ?o1 ;
bsbm:product>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:country>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:country>  ?o1 ;
dc:title>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:country>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:vendor>  ?o1 ;
bsbm:price>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:vendor>  ?o1 ;
bsbm:reviewFor>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:vendor>  ?o1 ;
bsbm:validTo>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:vendor>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:vendor>  ?o1 ;
bsbm:product>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:vendor>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:vendor>  ?o1 ;
dc:title>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:vendor>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:price>  ?o1 ;
bsbm:reviewFor>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:price>  ?o1 ;
bsbm:validTo>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:price>  ?o1 ;
dc:publisher>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:price>  ?o1 ;
bsbm:product>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:price>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:price>  ?o1 ;
dc:title>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:price>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:reviewFor>  ?o1 ;
bsbm:validTo>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:reviewFor>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:reviewFor>  ?o1 ;
bsbm:product>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:reviewFor>  ?o1 ;
rdfs:label>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:reviewFor>  ?o1 ;
dc:title>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:reviewFor>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:validTo>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:validTo>  ?o1 ;
bsbm:product>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:validTo>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:validTo>  ?o1 ;
dc:title>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:validTo>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  dc:publisher>  ?o1 ;
bsbm:product>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  dc:publisher>  ?o1 ;
dc:title>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  dc:publisher>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:product>  ?o1 ;
rdfs:label>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:product>  ?o1 ;
dc:title>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:product>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  rdfs:label>  ?o1 ;
dc:title>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  rdfs:label>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  dc:title>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
bsbm:reviewDate>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  foaf:name>  ?o1 ;
rev:text>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:reviewFor>  ?o1 ;
bsbm:reviewDate>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:reviewFor>  ?o1 ;
rev:text>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:reviewDate>  ?o1 ;
rev:text>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:reviewDate>  ?o1 ;
dc:title>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:reviewDate>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  rev:text>  ?o1 ;
dc:title>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  rev:text>  ?o1 ;
rev:reviewer>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:deliveryDays>  ?o1 ;
bsbm:country>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:deliveryDays>  ?o1 ;
bsbm:price>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:deliveryDays>  ?o1 ;
bsbm:vendor>  ?o2 .
}
```

Continued on next page

Table A.2 – continued from previous page

```
SELECT  *
WHERE
{ ?s  bsbm:deliveryDays>  ?o1 ;
bsbm:validTo>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:deliveryDays>  ?o1 ;
dc:publisher>  ?o2 .
}
```

```
SELECT  *
WHERE
{ ?s  bsbm:deliveryDays>  ?o1 ;
bsbm:product>  ?o2 .
}
```

Bibliography

M Acosta, ME Vidal, and T Lampo. ANAPSID: An adaptive query processing engine for SPARQL endpoints. In *proceedings of the International Semantic Web Conference (ISWC)*, pages 18–34, 2011. URL `http://www.springerlink.com/index/56475624X7744457.pdf`.

Ziya Akar, Tayfun Gökmen Halaç, and Erdem Eser Ekinci. Querying the web of interlinked datasets using VoID descriptions. In *proceedings of the Linked Data on the Web Workshop (LDOW) , at the International World Wide Web Conference (WWW)*, 2012. URL `http://events.linkeddata.org/ldow2012/papers/ldow2012-paper-06.pdf`.

Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing linked datasets on the design and usage of VoID , the "Vocabulary of Interlinked Datasets". In *proceedings of the Linked Data on the Web Workshop (LDOW) , at the International World Wide Web Conference (WWW)*, 2009.

Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. In *proceedings of the International Semantic Web Conference (ISWC)*, pages 114–129, 2008. URL `http://www.springerlink.com/index/7lv586v1j2j43156.pdf`.

S Bail, B Parsia, and U Sattler. JustBench: A framework for OWL benchmarking. In *proceedings of the International Semantic Web Conference (ISWC)*, pages 32–47, 2010. URL `http://link.springer.com/chapter/10.1007/978-3-642-17746-0_3`.

T Berners-Lee. Linked data-design issues, 2006. URL `http://www.w3.org/DesignIssues/LinkedData.html`.

T Berners-Lee, J Hendler, and O Lassila. The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, 2001.

Abraham Bernstein, Christoph Kiefer, and Markus Stocker. OptARQ : a SPARQL optimization approach based on triple pattern selectivity estimation. Technical report, Technical Report ifi-2007.03, Department of Informatics, University of Zurich, 2007.

Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 6(4):602–625, December 1981. ISSN 03625915. doi: 10.1145/319628.319650. URL `http://dl.acm.org/citation.cfm?id=319628.319650`.

Christian Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *International Journal On Semantic Web and Information Systems (IJSWIS) - Special Issue on Scalability and Performance of Semantic Web Systems*, 5(2):1–24, 2009.

Christoph Böhm, Johannes Lorey, and Felix Naumann. Creating voiD descriptions for Web-scale data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9(3):339–345, 2011. ISSN 1570-8268. URL `http://www.sciencedirect.com/science/article/pii/S1570826811000370`.

Peter Boncz, Minh-Duc Pham, Orri Erling, Ivan Mikhailov, and Yrjana Rankka. Social Network Intelligence BenchMark. URL `http://www.w3.org/wiki/Social_Network_Intelligence_BenchMark`.

Jeremy Carroll, Ivan Herman, and Peter F. Patel-Schneider. OWL 2 Web Ontology Language RDF-Based Semantics (Second Edition), 2012. URL `http://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/`.

Gong Cheng and Yuzhong Qu. Searching linked objects with Falcons: approach, implementation and evaluation. *International Journal on Semantic Web and Information Systems*, 5(3):49–70, 2009. ISSN 15526283. doi: 10.4018/jswis.2009081903. URL `http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/jswis.2009081903`.

S. Christodoulakis. Implications of certain assumptions in database performance evauation. *ACM Transactions on Database Systems*, 9(2):163–186, May 1984. ISSN

03625915. doi: 10.1145/329.318578. URL http://dl.acm.org/citation.cfm?id=329.318578.

Mathieu D'Aquin, Claudio Baldassarre, Laurian Gridinoc, Sofia Angeletou, Marta Sabou, and Enrico Motta. Characterizing knowledge on the semantic web with Watson. In *proceedings of the Evaluation of Ontologies and Ontology-Based Tools Workshop (EON), in conjuction with the International Semantic Web Conference (ISWC)*, volume 329, pages 1–10, 2007. URL http://oro.open.ac.uk/23555/.

David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992. ISSN 00010782. doi: 10.1145/129888.129894. URL http://dl.acm.org/citation.cfm?id=129888.129894.

Li Ding, Joshua Shinavier, Zhenning Shangguan, and Deborah McGuinness. SameAs networks and beyond: Analyzing deployment status and implications of owl: sameAs in linked data. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *proceedings of the International Semantic Web Conference (ISWC)*, volume 6496 of *Lecture Notes in Computer Science*, pages 145–160, Berlin, Heidelberg, 2010. ISBN 978-3-642-17745-3. doi: 10.1007/978-3-642-17746-0. URL http://www.springerlink.com/index/Q1359571L25472PK.pdf.

S Duan and A Kementsietsidis. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *proceedings of the ACM SIGMOD International Conference on Management of data (SIGMOD)*, 2011. URL http://dl.acm.org/citation.cfm?id=1989340.

G.H.L. Fletcher and P.W. Beck. Scalable indexing of RDF graphs for efficient join processing. In *proceeding of the 18th ACM conference on Information and knowledge management*, pages 1513–1516, 2009. URL http://portal.acm.org/citation.cfm?id=1646159.

Hugh Glaser, Tim Lewy, Ian Millard, and Ben Dowling. On coreference and the Semantic Web. In *proceedings of the European Semantic Web Conference (ESWC)*, 2007.

Hugh Glaser, Afraz Jaffri, and Ian Millard. Managing Co-reference on the Semantic web. In *proceedings of the Linked Data on the Web Workshop (LDOW) , at the International*

World Wide Web Conference (WWW), 2009. URL http://eprints.soton.ac.uk/267587/.

Olaf Görlitz and Steffen Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *proceedings of the Consuming Linked Data Workshop(COLD)*, 2011.

Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, June 1993. ISSN 03600300. doi: 10.1145/152610.152611. URL http://portal.acm.org/citation.cfm?id=152610.152611.

Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005. ISSN 1570-8268. URL http://linkinghub.elsevier.com/retrieve/pii/S1570826805000132.

Claudio Gutierrez. Foundations of RDF databases. *Reasoning Web. Semantic Technologies for Information Systems*, 5689:158–204, 2008. URL http://videolectures.net/site/normal_dl/tag=25799/eswc08_gutierrez_frdf_01.pdf.

L.M. Haas, D. Kossmann, E.L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *proceedings of the International Conference on Very Large Data Bases*, pages 276–285, 1997. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.7606&rep=rep1&type=pdf.

H Halpin and PJ Hayes. When owl: sameAs isn't the same: An analysis of identity links on the semantic web. *proceedings of the Linked Data on the Web Workshop (LDOW) , at the International World Wide Web Conference (WWW)*, 2010. URL http://events.linkeddata.org/ldow2010/papers/ldow2010_paper09.pdf.

Andreas Harth and S. Decker. Optimized index structures for querying rdf from the web. In *proceedings of the Third Latin American Web Congress (LA-WEB)*, page 10, 2006. ISBN 0769524710. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1592360.

Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, K.U. Sattler, and Jürgen Umbrich. Data summaries for on-demand queries over linked data. In *Proceedings of the 19th international conference on World wide web*, pages 411–420, New York,

New York, USA, 2010. ISBN 9781605587998. doi: 10.1145/1772690.1772733. URL http://portal.acm.org/citation.cfm?id=1772733.

Olaf Hartig. Zero-Knowledge query planning for an iterator implementation of link traversal based query execution. In Grigoris Antoniou, Marko Grobelnik, Elena Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Pan, editors, *proceedings of the European Semantic Web Conference (ESWC)*, volume 6643 of *ESWC'11*, pages 154–169, 2011. ISBN 9783642210334. URL http://dx.doi.org/10.1007/978-3-642-21034-1_11.

Olaf Hartig and Christian Bizer. Executing SPARQL Queries over the Web of Linked Data. *The Semantic Web-ISWC 2009*, 5823:293–309, 2009. URL http://www.springerlink.com/index/Q37381173G66W7N2.pdf.

Patrick Hayes and Brian McBride. RDF semantics, 2004. URL http://www.w3.org/TR/rdf-mt/.

Tom Heath and Christian Bizer. Linked Data: Evolving the Web into a global data space. *Synthesis Lectures on the Semantic Web: Theory and Technology*, 1(1):1–136, February 2011. ISSN 2160-4711. doi: 10.2200/S00334ED1V01Y201102WBE001. URL http://www.citeulike.org/user/mikel_egana/article/10524056.

Aidan Hogan, Andreas Harth, and S Decker. Performing object consolidation on the semantic web data graph. In *proceedings of 1st I3: Identity, Identifiers, Identification Workshop*, 2007. URL http://aran.library.nuigalway.ie/xmlui/handle/10379/493.

Aidan Hogan, Andreas Harth, Jürgen Umbrich, Sheila Kinsella, Axel Polleres, and Stefan Decker. Searching and browsing Linked Data with SWSE: the Semantic Web search engine. *Semantic Search over the Web*, pages 361–414, 2012. ISSN 15708268. doi: 10.1016/j.websem.2011.06.004. URL http://linkinghub.elsevier.com/retrieve/pii/S1570826811000473.

W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, (1):9–32, 1993. doi: 10.1109/PDIS.1991.183106.

Wei Hu, Jianfeng Chen, and Yuzhong Qu. A self-training approach for resolving object coreference on the semantic web. In *proceedings of the International Conference on*

World Wide Web (WWW), page 87, March 2011a. ISBN 9781450306324. doi: 10.1145/1963405.1963421. URL http://dl.acm.org/citation.cfm?id=1963405.1963421.

Wei Hu, Jianfeng Chen, Hang Zhang, and Yuzhong Qu. How matchable are four thousand ontologies on the semantic Web. *The Semantic Web: Research and Applications*, 6643:290–304, 2011b. doi: 10.1007/978-3-642-21034-1. URL http://www.springerlink.com/index/259681U010372545.pdf.

Bernadette Hyland, Boris Villazón-Terrazas, and Ghislain Atemezing. Best practices for publishing Linked Data (W3C editor's draft 13 March 2013), 2013. URL https://dvcs.w3.org/hg/gld/raw-file/default/bp/index.html.

Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An adaptive query execution system for data integration. *ACM SIGMOD Record*, 28 (2):299–310, June 1999. ISSN 01635808. doi: 10.1145/304181.304209. URL http://dl.acm.org/citation.cfm?id=304181.304209.

Afraz Jaffri, Hugh Glaser, and Ian Millard. Uri disambiguation in the context of linked data. In *proceedings of the Linked Data on the Web Workshop (LDOW) , at the International World Wide Web Conference (WWW)*, 2008. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.9313&rep=rep1&type=pdf.

G. Klyne, J.J. Carroll, and B. McBride. Resource description framework (RDF): Concepts and abstract syntax, 2004. URL http://www.w3.org/TR/rdf-concepts/.

Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000. ISSN 0360-0300. URL http://portal.acm.org/citation.cfm?id=371598&dl=.

Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems*, 25(1):43–82, March 2000. ISSN 03625915. doi: 10.1145/352958.352982. URL http://doi.acm.org/10.1145/352958.352982.

G. Ladwig and Thanh Tran. Linked Data Query Processing Strategies. *The Semantic Web-ISWC 2010*, pages 453–469, 2010. URL http://www.springerlink.com/index/Q6385N8J0071156U.pdf.

Günter Ladwig and Thanh Tran. SIHJoin: Querying remote and local Linked Data. *The Semantic Web: Research and Applications*, 6643:139–153, 2011. doi: 10.1007/978-3-642-21034-1. URL http://www.springerlink.com/content/d7v4716326776w7l/.

Andreas Langegger and Wolfram Woss. RDFStats - An extensible RDF statistics generator and library. In *proceedings of the International Workshop on Database and Expert Systems Application*, pages 79–83, August 2009. ISBN 978-0-7695-3763-4. doi: 10.1109/DEXA.2009.25. URL http://www.computer.org/portal/web/csdl/doi/10.1109/DEXA.2009.25.

Andreas Langegger, Wolfram Wöß, and Martin Blöchl. A Semantic Web middleware for virtual data integration on the Web. *The Semantic Web Research and Applications*, 5021:493–507, 2008. ISSN 03029743. doi: 10.1007/978-3-540-68234-9_37. URL http://www.springerlink.com/index/1804822517684043.pdf.

Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. *ACM SIGMOD Record*, 17(3):18–27, June 1988. ISSN 01635808. doi: 10.1145/971701.50204. URL http://dl.acm.org/citation.cfm?id=971701.50204.

L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. Towards a complete OWL ontology benchmark. *The Semantic Web: Research and Applications*, 4011:125–139, 2006. URL http://www.springerlink.com/index/l0wu543x26350462.pdf.

LF Mackert. R* optimizer validation and performance evaluation for distributed queries. *Proceedings of the 1986 ACM SIGMOD international conference on Management of data (SIGMOD '86)*, pages 149–159, August 1988. URL http://dl.acm.org/citation.cfm?id=645913.671480http://books.google.com/books?hl=en&lr=&id=7a48qSMuVcUC&oi=fnd&pg=PA175&dq=R*+optimizer+validation+and+performance+evaluation+for+distributed+queries&ots=t9pg6zRm4f&sig=vDISxHaqMECEkY67QlkcGgZm7ZU.

Priti Mishra and Margaret H. M.H. Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, January 1992. ISSN 0360-0300. doi: 10.1001/archoto.2007.12. URL http://portal.acm.org/citation.cfm?id=128764&dl=http://dl.acm.org/citation.cfm?id=128762.128764.

M Morsey, J Lehmann, S Auer, and ACN Ngomo. DBpedia SPARQL benchmark-performance assessment with real queries on real data. *The Semantic Web-ISWC 2011*, 2011. URL http://link.springer.com/chapter/10.1007/978-3-642-25073-6_29.

Thomas Neumann. RDF-3X: a RISC-style engine for RDF. *proceedings of the VLDB Endowment*, pages 647–659, 2008. URL http://portal.acm.org/citation.cfm?id=1453856.1453927.

Eyal Oren, R Delbru, M Catasta, R Cyganiak, H. Stenzhorn, and G. Tummarello. Sindice. com: A document-oriented lookup index for open linked data. *International Journal of Metadata, Semantics and Ontologies*, 3(1):37–52, 2008. URL http://inderscience.metapress.com/index/3518208222365647.pdf.

MT Özsu and P. Valduriez. *Principles of distributed database systems*. 1999. ISBN 8177581775. URL http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Principles+of+distributed+database+systems#0.

J Pérez and Marcelo Arenas. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 2009. URL http://portal.acm.org/citation.cfm?id=1567274.1567278.

MD Pham, P Boncz, and O Erling. S3G2: a scalable structure-correlated social graph generator. *Selected Topics in Performance Evaluation and Benchmarking*, 7755:156–172, 2013. URL http://link.springer.com/chapter/10.1007/978-3-642-36727-4_11.

Viswanath Poosala and Yannis E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 486–495, August 1997. ISBN 1-55860-470-7. URL http://dl.acm.org/citation.cfm?id=645923.673638.

R.C. Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957. URL http://orion.research.bell-labs.com/BSTJ/images/Vol36/bstj36-6-1389.pdf.

Eric Prud'hommeaux and C Buil-Aranda. SPARQL 1.1 federated query, 2013. URL http://www.w3.org/TR/sparql11-federated-query/.

Eric Prud'Hommeaux and Andy Seaborne. SPARQL query language for RDF, 2008. URL http://www.w3.org/TR/rdf-sparql-query/.

M Przyjaciel-Zablocki, A Schätzle, T Hornung, and I Taxidou. Towards a SPARQL 1.1 Feature Benchmark on Real-World Social Network Data. In *proceedings of the 1st International Workhop on Benchmarking RDF Systems*, 2013. URL http://ceur-ws.org/Vol-981/BeRSys2013paper1.pdf.

Bastian Quilitz. Querying distributed RDF data sources with SPARQL. *The Semantic Web: Research and Applications*, pages 524–538, 2008. URL http://www.springerlink.com/index/hm1v15q75371640p.pdf.

Louiqa Raschid and Stanley Y. W. Su. A parallel processing strategy for evaluating recursive queries. In *proceedings of the 12th International Conference on Very Large Data Bases (VLDB)*, pages 412–419, August 1986. ISBN 0-934613-18-4. URL http://dl.acm.org/citation.cfm?id=645913.671471.

Simon Schenk and S. Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In *proceeding of the International Conference on World Wide Web (WWW)*, pages 585–594, 2008. URL http://portal.acm.org/citation.cfm?id=1367497.1367577.

Simon Schenk, Carsten Saathoff, and Steffen Staab. SemaPlorer-Interactive semantic exploration of data and media based on a federated cloud infrastructure. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(4):298–304, 2009.

M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A benchmark suite for federated semantic data query processing. *ISWC 2011*, 2011. URL http://www.informatik.uni-freiburg.de/~mschmidt/docs/iswc11_fedbench.pdf.

Michael Schmidt, Thomas Hornung, Georg Lausen, and C. Pinkel. SP2Bench: A SPARQL performance benchmark. In *proceedings of the International Conference on Data Engineering*, pages 222–233, 2009. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4812405.

A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *proceedings of the*

International Semantic Web Conference (ISWC), 2011. URL http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Research_Paper/05/70310592.pdf.

P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, page 23, New York, New York, USA, May 1979. ISBN 089791001X. doi: 10.1145/582095.582099. URL http://portal.acm.org/citation.cfm?id=582095.582099.

Steffen Staab. Federated data management and query optimization for Linked Open Data. *New Directions in Web Data Management 1*, 331:109–137, 2011. doi: 10.1007/978-3-642-17551-0_5. URL http://www.springerlink.com/index/B2470QJ11563Q242.pdf.

Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The International Journal on Very Large Data Bases*, 6(3):191–208, August 1997. ISSN 1066-8888. doi: 10.1007/s007780050040. URL http://portal.acm.org/citation.cfm?id=765554.765556.

Markus Stocker, Andy Seaborne, Abraham Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *proceeding of the International Conference on World Wide Web (WWW)*, pages 595–604, 2008. URL http://portal.acm.org/citation.cfm?id=1367497.1367578.

Heiner Stuckenschmidt, Richard Vdovjak, G.J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed RDF repositories. In *proceedings of the International Conference on World Wide Web (WWW)*, pages 631–639, 2004. URL http://portal.acm.org/citation.cfm?id=988758.

J Umbrich, A Hogan, A Polleres, and S Decker. Improving the Recall of Live Linked Data Querying through Reasoning. *Web Reasoning and Rule Systems*, 7497:188–204, 2012. doi: 10.1007/978-3-642-33203-6. URL http://www.springerlink.com/index/10.1007/978-3-642-33203-6.

T Urhan and MJ Franklin. XJoin: Getting fast answers from slow and bursty networks. *University of Maryland Technical Report CS-TR-3994 (Feb.)*, 1999.

Ben P. Vandervalk, E. Luke McCarthy, and Mark D. Wilkinson. Optimization of Distributed SPARQL Queries Using Edmonds' Algorithm and Prim's Algorithm.

In *proceedings of the International Conference on Computational Science and Engineering*, volume 1, pages 330–337, 2009. doi: 10.1109/CSE.2009.144. URL http://www.computer.org/portal/web/csdl/doi/10.1109/CSE.2009.144.

Xin Wang, Thanassis Tiropanis, and Hugh C. Davis. Evaluating graph traversal algorithms for distributed SPARQL query optimization. In *proceedings of the Joint International Semantic Technology Conference (JIST)*, 2011.

Xin Wang, Thanassis Tiropanis, and Hugh C. Davis. LHD: Optimising Linked Data query processing using parallelisation. In *proceedings of the Linked Data on the Web Workshop (LDOW) , at the International World Wide Web Conference (WWW)*, 2013.

Gregory Todd Williams and Rensselaer Polytechnic Institute. SPARQL 1.1 Service Description. W3C Working Draft (12 May 2011), 2011.

AN Wilschut and PMG Apers. Dataflow query execution in a parallel main-memory environment. In *proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 68–77, December 1993. ISBN 0-8186-2295-4. URL http://dl.acm.org/citation.cfm?id=382009.383658http://www.springerlink.com/index/R2274PH377725185.pdf.

DH Wolpert and WG Macready. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010 (Santa Fe Institute), 1995. URL http://delta.cs.cinvestav.mx/~ccoello/compevol/nfl.pdf.

DH Wolpert and WG Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=585893.