Provided by the author(s) and University of Galway in accordance with publisher policies. Please cite the published version when available.

Title	Evaluating and benchmarking the performance of federated SPARQL endpoints and their partitioning using selected metrics and specific query types
Author(s)	Rakhmawati, Nur Aini
Publication Date	2017-02-07
Item record	http://hdl.handle.net/10379/6284

Downloaded 2024-04-25T21:30:27Z

Some rights reserved. For more information, please see the item record link above.

Evaluating and benchmarking the performance of federated SPARQL endpoints and their partitioning using selected metrics and specific query types

Nur Aini Rakhmawati

Supervisor: **Prof. Stefan Decker**
Supervisor: **Prof. Dietrich Rebholz-Schuhmann**
Internal Examiner: **Dr. John Breslin**
External Examiner: **Prof. Sören Auer**

Dissertation submitted in pursuance of the degree of Doctor of Philosophy

The Insight Centre for Data Analytics, Galway
National University of Ireland, Galway / Ollscoil na hÉireann, Gaillimh

February 8, 2017

Copyright © Nur Aini Rakhmawati, 2016

The research presented herein was supported by an Indonesian Directorate General of Higher Education Scholarship and an IRCSET Postgraduate Scholarship by Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289

"If you have an apple and I have an apple and we exchange these apples then you and I will still each have one apple. But if you have an idea and I have an idea and we exchange these ideas, then each of us will have two ideas."

—George Bernard Shaw

Acknowledgements

First, thanks to Allah;

...thanks to my family, particularly my hubby, Key, Irelanda, my Mum, my Dad and Bu Nina;

...thanks to Stefan Decker and Dietrich Rebholz-Schuhmann for the guidance;

...thanks to the various students and staff of DERI;

...thanks to the LIDRC folk, particularly Michael, Richard, Myr and Aftab :);

...thanks to John Breslin, Sören Auer and Conor Hayes

...thanks to people with whom I have worked *very* closely, particularly Ali, Oya, Marcel, Soheila, Saleem, Sarasi;

...thanks to Indonesian people who are living in Galway;

Abstract

The increasing amount of Linked Data and its inherent distributed nature have created for need to developing and researching querying technologies. Inspired by research results from traditional distributed databases, different approaches for managing federation over SPARQL Endpoints have been introduced. Such a system consists of a federated engine as the query mediator and a group of SPARQL endpoints as the data provider. SPARQL is the standardised query language for RDF, the default data model used in Linked Data deployments, and SPARQL endpoints are a popular access mechanism provided by many RDF repositories.

The growth of the number of federated SPARQL query systems creates the necessity for benchmarking systems to evaluate their performance. Designing a benchmark for a federated SPARQL query system is a non-trivial task since it consists of heterogeneous systems (e.g. hardware, software, data structure and data distribution) which are also distributed. In this thesis, we design a comprehensive benchmark based on the dependencies between the metrics, datasets and queries. We initially investigate existing federated engines and compare their features and behaviours. Based on this investigation, we first identify the metrics that are suitable to assess the performance of federated SPARQL query systems. We introduce three types of metrics: independent metrics, semi-independent metrics and composite metrics. Thereafter, we investigate the benefits and the costs associated while federating a SPARQL query over multiple sources having links between them in the existing federated engines. Next, we present six approaches to generate a dataset for benchmarking a federated SPARQL queries. Thereafter, by using those approaches, we generate 9 datasets and then observe the relationship between the spreading factor of those datasets and the communication cost. The spreading factor is a dataset metric for computing the distribution of classes and properties throughout a set of data sources. Finally, we present QFed, a dynamic SPARQL query set generator for federated SPARQL query benchmarks that takes into account the characteristics of both datasets and queries along with the metrics.

Declaration

I declare that this thesis is composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Nur Aini Rakhmawati
February 8, 2017

Contents

1 Introduction **1**
1.1 Problem Statement 6
1.1.1 Challenges 6
1.1.2 Research Question 7
1.2 Hypothesis 11
1.3 Contribution and Thesis Structure 12
1.4 Impact 14
1.5 Thesis Scope 15

2 Background **16**
2.1 Semantic Web 16
2.1.1 Resource Description Format(RDF) 16
2.1.2 SPARQL 18
2.2 Linked Data 25
2.2.1 Querying over Linked Data Infrastructure 25
2.2.2 Centralised Repository 26
2.2.3 Distributed Repositories 26

3 Querying over Federated SPARQL Endpoints —A survey of the State of the Art **30**
3.1 Federation Feature in SPARQL 1.1 30
3.2 Architecture of Federated SPARQL Query Engines 32
3.3 Federated SPARQL Query Approaches 33
3.3.1 Using a popular predicate and a variable in the predicate position . . 35
3.3.2 Using a `UNION` Operator 35
3.3.3 Object similarity 35
3.3.4 Using Links 36
3.4 Basic Steps of Query Process in a Federated Engine 37
3.4.1 Query Parsing 37
3.4.2 Source Selection 37
3.4.3 Source Tracking 40
3.4.4 Query Planning 41
3.4.5 Query Execution 41

3.5 The Existing Frameworks for Federation over SPARQL Endpoints 42
3.5.1 Rewriter architectures 42
3.5.2 Planner Architectures 44
3.6 Challenges 48
3.7 Conclusion 50

4 A Proposal for New Performance Metrics for Query Federation Benchmarks 52
4.1 State of the art of RDF repository benchmarks 52
4.1.1 Benchmark for a single RDF repository 53
4.1.2 Distributed RDF Repositories 54
4.2 Proposed Performance Metrics 55
4.2.1 Basic Metrics 55
4.2.2 Extended Metrics 57
4.3 Experiment and Result 65
4.4 Conclusion 69

5 The Cost And Benefit of Exploiting Links in Federated SPARQL Query 71
5.1 Motivating Example 71
5.2 Related Work 73
5.3 Interlinking in Federation over SPARQL Endpoints 73
5.4 The Cost and Benefit of the Links 76
5.5 Experimental Setup 78
5.6 Results and Discussion 81
5.7 Conclusion 85

6 Dataset for Query Federation Benchmark 86
6.1 The states of the art of RDF data partitions 86
6.1.1 METIS Partition 86
6.1.2 Vertically Partitioned Tables 88
6.1.3 Property Tables 88
6.1.4 Class Partition 90
6.1.5 Hash Partition 90
6.1.6 Discussion 90
6.2 Dataset Generation 90
6.2.1 METIS Partition 91
6.2.2 Entity Partition 93
6.2.3 Class Partition 94
6.2.4 Property Partition 94
6.2.5 Triples Partition 95
6.2.6 Hybrid Partition 95
6.3 DFedQ 96

7 On Metrics For Measuring Fragmentation Of Federation over SPARQL Endpoints **97**
7.1 Motivating Example 97
7.1.1 Spreading Factor of a Dataset 99
7.1.2 Spreading Factor of a Dataset associated with the Query set 103
7.2 Evaluation 104
7.2.1 System 105
7.2.2 Query set 105
7.2.3 Dataset 106
7.2.4 Metrics 109
7.3 Results and discussion 109
7.4 Conclusion 111

8 QFed: Query Benchmark Generator Based on Metrics and Characteristics of a dataset **113**
8.1 Query Requirements For Federation Framework Benchmark 114
8.2 Query Set Generation 115
8.2.1 Methodology 115
8.2.2 Data Preprocessing 116
8.2.3 Query Join Pattern Types 117
8.2.4 Star shape Creation 125
8.2.5 Queryset Generation Extension 130
8.2.6 Comparison of Splodge, Lidaq and QFed 131
8.3 Evaluation 132
8.3.1 Experimental Setup 132
8.3.2 Results and Discussion 134
8.4 Conclusion 138

9 Discussion and Conclusions **139**
9.1 Future Directions 141

A Prefixes **156**

B Query Set **157**
B.1 Dailymed Queries 157
B.2 Link and No Link Queries 160
B.3 QFed Queries 163

C Dataset **166**

D The accuracy results **168**

Chapter 1

Introduction

The Resource Description Framework (RDF)[1] is a World Wide Web Consortium (W3C)[2] standard for exchanging data in the Web which allows machines seamlessly to process information available on the web. At present, a huge amount of data has been converted to RDF and more and more data have been annotated with RDF. The SPARQL Protocol and RDF Query Language (SPARQL)[3] was officially introduced in 2008 to retrieve RDF data similar to how the SQL[4] query language is used to query relational databases. As the Web of data grows and more and more applications are developed to exploit the rich information, there is also growth in the number of SPARQL endpoints constructing and running the SPARQL queries over the Web of Data using HTTP. According to Linked Open Data cloud statistics[5], 68.14% of public RDF repositories are equipped with a SPARQL endpoint, SPARQL is becoming de facto standard for accessing RDF data, as it provides a flexible way to interact with the Web of Data by formulating a query like SQL in relational databases. In addition, a SPARQL endpoint can return the query answers in several formats, such as XML[6] and JSON[7] which are widely used as data exchange standards in various applications. But, aggregating data from multiple SPARQL endpoints remains a challenge which received increased interest in recent years. Thus, we investigate several factors that are related to a benchmark various engines that merge data from multiple SPARQL endpoints. After that, we propose several approaches to benchmark those engines.

[1]RDF: `http://www.w3.org/RDF/`
[2]W3C: `http://www.w3.org`
[3]SPARQL: `http://www.w3.org/TR/rdf-sparql-query/`
[4]SQL: `http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498`
[5]Linked Open Data: `http://lod-cloud.net/state/`
[6]XML: `http://www.w3.org/XML/`
[7]JSON: `http://www.json.org`

To date, distributed RDF repositories applications have been actively developed mostly in the life science domain since many life science datasets are available on the LOD cloud[8] and can be accessed publicly. At the moment, there are more than 40 life science public datasets which are publicly available on the LOD cloud. The Health Care and Life Sciences (HCLS) domain advocated Linked Data from its early days, and currently a considerable portion of the Linked Data cloud consists of datasets from the Linked Data for Life Sciences (LD4LS) domain [Hasnain et al., 2012]. Currently, there are multiple datasets from HCLS projects, including *bio2rdf*[9], the Health Care and *Life Sciences Knowledge base*[10] (HCLS Kb), *linkedlifedata*[11], Linked Open Drug Data effort[12] and the *Neurocommons*[13]. These efforts are motivated by and have originated from biomedical facilities in recent years, partially caused by the decrease cost of acquiring large datasets such as *genomics* sequences as well as the trend towards personalised medicine, *pharmacogenomics* and integrative Bioinformatics, which require accessing and querying life sciences data. [Hernandez and Kambhampati, 2004] described the high demand for biological dataset integration to support life science researcher. Although the publication of datasets as RDF is a significant milestone towards the feasibility of querying these healthcare and other biological datasets, to date, creating a query-able Web of HCLS data is not a trivial task.

In order to get a better understanding of the need of federated SPARQL endpoints, consider two SPARQL endpoints DrugBank[14] and Kegg[15] (Figure 1.1) that publish information regarding drugs and compounds. Both the datasets have different useful information about the same concepts. The oval in Figure 1.1 represents an entity, whereas the box denotes the value of property. Querying data from three datasets produces drug information such as a drug's indication and compounds. In order to answer this question *Find the chemical equations and reaction titles of reactions related to only those drugs which are approved along with their average Molecular Weight*, we should send the query to both SPARQL endpoints. Kegg contains two concepts, namely *Chemical equations* and *Reaction title* whereas the information like average *Molecular Weight* and *approved drugs* are found in the Drugbank

[8]The LOD (Linked Open Data) Cloud is consists of datasets from multiple domains where DBPedia as a central hub. The domains include life science, geographic data, media, government, publication as well as cross domain: `http://lod-cloud.net`

[9]`http://bio2rdf.org/`

[10]`http://www.w3.org/TR/hcls-kb/`

[11]`http://linkedlifedata.com/`

[12]`http://www.w3.org/wiki/HCLSIG/LODD`

[13]`http://neurocommons.org/page/Main_Page`

[14]Drugbank: `http://wifo5-03.informatik.uni-mannhem.de/drugbank/`

[15]Kegg: `http://s4.semanticscience.org:16036/sparql`

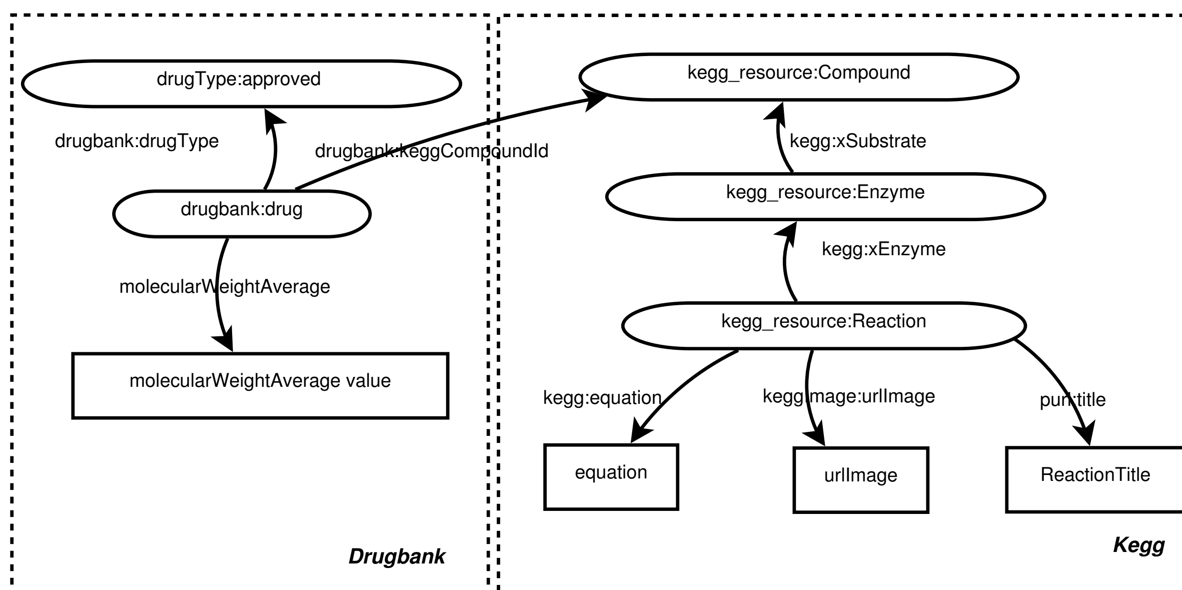


Figure 1.1: Example data relation from the Drugbank and the Kegg datasets

SPARQL endpoint.

A noteworthy application of distributed RDF repositories is presented in [Cheung et al., 2009] primarily for supporting neuroscience research. NeuroWiki[16] also collects neuroscience data from multiple life science RDF stores by using Linked Data Integration Framework (LDIF) [Schultz et al., 2012]. Two interesting works on query federation on Cancer disease were proposed by [Saleem et al., 2013c; Gonzälez-Beltrn et al., 2012]. Apart from the implementation of federated queries in the life science domain, a query over distributed RDF datasets system has also been applied to collect research publication information from more than 20 datasets under the `rkbexplorer.com` domain [Millard et al., 2010].

Due to a growing demand for applications that collect and exploit data from multiple datasets, federated engines have been actively developed to facilitate data integration amongst SPARQL endpoints [Rakhmawati et al., 2013]. These federated engines include DARQ [Quilitz and Leser, 2008], SemWiq [Langegger et al., 2008], Splendid [Görlitz and Staab, 2011], FedX [Schwarte et al., 2011], Anapsid [Montoya et al., 2012a] and Avalanche [Basca and Bernstein, 2010]. Therefore, there is a need for benchmarking the existing federated engines as a main part of a federated SPARQL query framework. A benchmark system helps to compare the existing federated engines and eventually the results of benchmarking will help us to choose a suitable federated engine for any semantic web application that depends on federation. A federated SPARQL query framework is a

[16]NeuroWiki: `http://neurowiki.alleninstitute.org/`

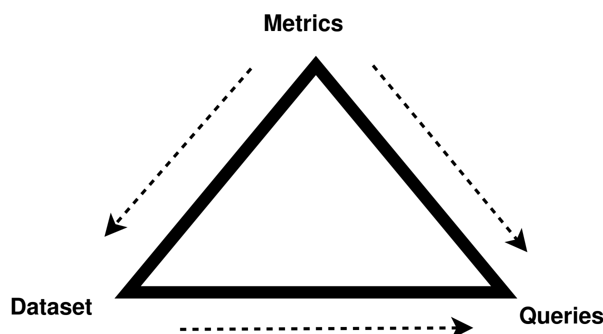


Figure 1.2: Dependencies between metrics, datasets and queries

system that consists of a *federated engine* as the query mediator and a group of *SPARQL endpoints* as data providers. The federated engine has several important roles, including: distributing a query from the clients to SPARQL endpoints, merging the answers from the SPARQL endpoints and then returning the results back to the clients.

Benchmarking is a process to compare a number of systems based on specific indicators of performance. The outcomes of benchmarking are meant to help improving existing systems and help to choose the best system for the next implementation. In general, the evaluations of the performance of a single RDF store and federated RDF stores borrow ideas from the field of relational database benchmarking [Guo et al., 2007]. In terms of database benchmarking, the benchmark is usually comprised of three main components: 1) metrics 2) a dataset and 3) queries. In a nutshell, the relationship between these components of a federated engine can be described in Figure 1.2, where arrows indicate how each component influences the other components. Typically, the metrics component influences the design and selection of the dataset and queries. For instance, if we consider runtime as a performance indicator, then the complexity of the query should be one of the parameters considered when designing a query benchmark. If the cost of data communication is the performance indicator considered, then the query should be designed to retrieve more results. Also, the dataset design should consider the number of data sources within the federation framework, the data distribution, etc [Rakhmawati and Hausenblas, 2012]. The dataset also influences the generation of queries, in other words the knowledge about the characteristics of the dataset is necessary in order to generate queries that evaluate specific properties of the federated SPARQL query system [Görlitz et al., 2012].

FedBench, which is the only benchmark suite for federated SPARQL queries, is useful to test general purpose federated SPARQL query systems, but it cannot be used for the performance evaluation of specialised federation systems. This is because the benchmark is

comprised of predefined static queries and a set (singleton) of data sources which may be completely irrelevant to the specialised federation system in hand. For example, TopFed [Saleem et al., 2014, 2013a] is a specialised federation system, particularly designed for the Linked Cancer Genome Atlas (LTCGA[17]) [Saleem et al., 2013a] data set. However, to the best of our knowledge, none of the existing SPARQL query federation benchmarks include the LTCGA data set and queries. Similarly, SAFE[18] is developed for SPARQL query federation over linked statistical data using the RDF DataCube vocabulary[19] and thus it is essential to test this system with RDF DataCube data sets and corresponding SPARQL queries. In a nutshell, to test such specialised systems, we need a dynamic query generator which can generate a variety of federated SPARQL queries over the given set of data sources. Apart from that, the existing evaluations for assessing the federation over SPARQL endpoints [Montoya et al., 2012c; Schwarte et al., 2012] usually run their experiment over different datasets and different query sets. In fact, the performance of the federated engine is influenced by both the dataset and the query set. As a result, the performance results may vary. For benchmarking, a better comparison of federated engines performance can make with either static query sets over different datasets or static datasets with various query sets. Prasser [Prasser et al., 2012] have implemented three partitions for evaluating a distributed RDF repositories system: naturally-partitioned, horizontally-partitioned and randomly-partitioned. Comparing values of a specific performance metric across different data distributions is particularly useful for investigating which characteristics of a dataset that influence the performance of a federated engine. Moreover, running an evaluation over different datasets can show the flexibility of an engine for various datasets. Therefore, in this thesis, we propose queries and a dataset generator for benchmarking federated SPARQL queries. With respect to metrics, the existing evaluations use metrics that only assess the performance of a federated engine regardless of the existence of SPARQL endpoints. In fact, the SPARQL endpoints contribute to the performance of a federated engine. Thus, we investigate performance metrics that assess both the federated engine and the SPARQL endpoints[20]

[17]Linked TCGA: `http://tcga.deri.ie/`

[18]SAFE: `http://linked2safety.hcls.deri.org:8080/SAFE-Demo/`

[19]RDF DataCube: `http://www.w3.org/TR/vocab-data-cube/`

[20]Please note that the federated engines could be assessed from a variety of viewpoints, e.g., completeness, accuracy. In this thesis we focus on performance, we assume that query completeness and accuracy have a high dependency on SPARQL endpoint capacity and configuration (The detail explanation can be found at page 10)

1.1 Problem Statement

1.1.1 Challenges

1. **Multiple Locations**

   Federation over SPARQL endpoints has many similar characteristics to distributed relational databases as in both cases the data are distributed in multiple locations. Communication between machines is required for executing a query. As a result, the network can influence the performance of federated SPARQL query engines. The network location of SPARQL endpoints will contribute to the performance of a federated engine. For instance, the response time of a SPARQL endpoint that is located far away from the federated engine is slower than others. Eventually, the execution time of the federated engine takes longer.

2. **Heterogeneous Systems**

   Another challenge related to accessing datasets from multiple locations, as shown in Figure 1.3, is that the system can consist of heterogeneous components in terms of the hardware, software and dataset characteristics. For example, the federated engine runs on a machine that has high hardware specifications, while the SPARQL endpoints are set-up on the machines with low hardware specifications. Consequently, this slow query execution in a federated engine since the federated engine must wait for the SPARQL endpoints to execute a subquery. Furthermore, each SPARQL endpoint may use different underlying software that has different capabilities for managing RDF data. Apart from that, each SPARQL endpoint stores datasets with different characteristics such as the dataset size, dataset structure, etc.

3. **An Holistic Benchmark**

   Each component in a federated system can contribute to the performance of the whole federation framework. Thus, it is challenging to evaluate the federation framework performance. Ideally, we would like to benchmark that evaluates both the federated engine and SPARQL endpoints in a holistic way. Then different federated engines could be better assessed and compared, but most current evaluation approaches disregard the existence of the SPARQL endpoint server. Hence, the existing approaches for query optimisation in federated engines only focus on getting results in a timely manner without considering the capability of the SPARQL endpoint.

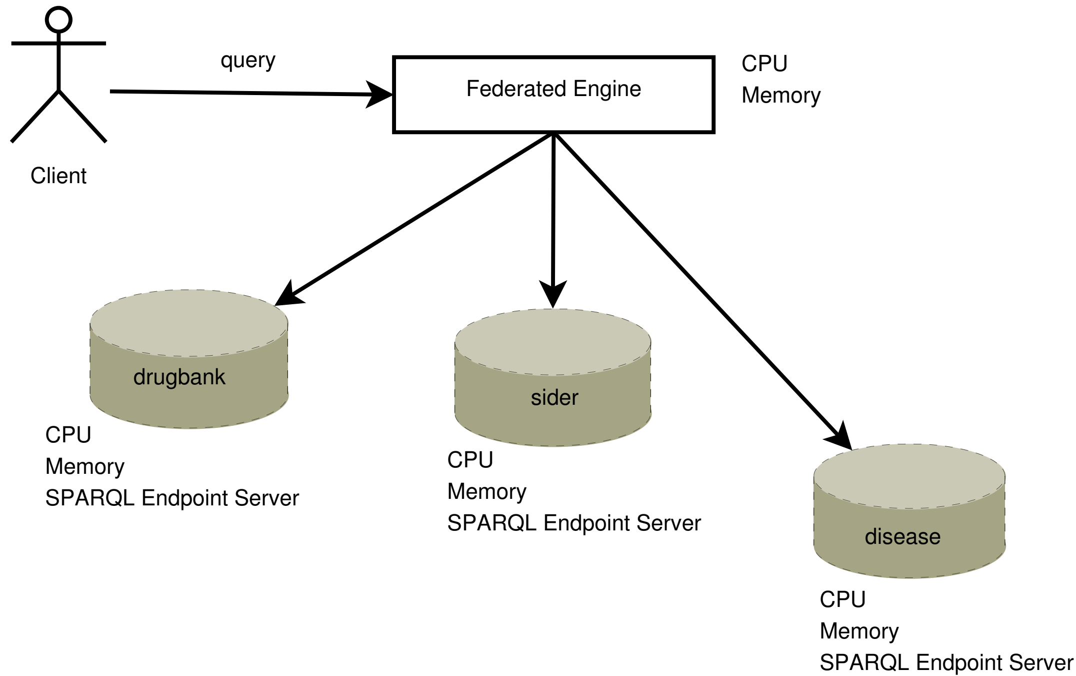


Figure 1.3: Heterogeneity in federated SPARQL endpoints framework

1.1.2 Research Question

Considering the challenges discussed in the previous section, this thesis addresses the following research question:

> **How can we design a comprehensive SPARQL query federation benchmark that considers the dependencies between metrics, dataset and queries?**

The research question can be broken down as follows:

- *Q1 : What are suitable metrics for assessing the performance of a federated SPARQL query framework?*
  - **A metric should not depend on the hardware environment.**
    A benchmark should produce the same results in a range of hardware environments (i.e. small and large scale systems) if the federated engine, SPARQL endpoints, queries and datasets are the same as the ones used in previous evaluations. It is important to establish a gold standard when people re-run a similar evaluation from the prior conducted evaluation. The gold standard can illustrate what

the next evaluation result will look like. If the next evaluation produces a result that is not in the range of the gold standard, this could be an indication that something went wrong while re-running the evaluation. Possible issues could include an incompatible system issue, a wrong configuration, etc. Therefore, a benchmark should be widely examined and approved by the community that needs it. The community contributes to fostering benchmark quality by testing the benchmark and sharing the benchmark results. Such benchmarks already exist in the database community since the database field is more mature than the semantic web field. A notable database benchmark is TPC[21] where numerous vendors contribute by evaluating their system and by sharing the evaluation configuration and results, with the objective of verifying and standardising the evaluation process. In this way, a researcher cannot claim that their system can surpass other systems without verification. The federated SPARQL query field is still far away from this kind of maturity. To the best of our knowledge, the Linked Data Benchmark Council (LDBC) [Boncz, 2013] is the only Linked Data benchmark that adopts the TPC style. Nevertheless, as of this writing, LDBC is still in the early stages of its development and does not handle federated SPARQL queries. Thus, we investigate a set of independent metrics for assessing the performance of the query federation framework.

– **A metric should assess both the federated engine and SPARQL endpoints.**

  In light of the above, the performance of the federated SPARQL query framework is influenced by the federated engine and a group of SPARQL endpoints. The capabilities of the SPARQL endpoint server while answering a query (e.g how quickly it executes a query, how many queries can be processed in a certain interval time, how many rows can be returned as a query answer) determine the final results that are delivered to the client. In order to obtain a meaningful summary of the evaluation, a metric should assess both the federated engine and the SPARQL endpoints. In this way, eventually, we can investigate which factors contribute the most to the query execution.

- *Q2 : Which characteristics of a dataset influence the performance of the query federation framework?*

[21]TPC: `http://www.tpc.org`

A federated SPARQL query framework consists of a group of autonomous SPARQL endpoints which handle different dataset characteristics. The dataset characteristics, particularly influence the source selection stage because most federated engines select relevant SPARQL endpoints that potentially answer the query before executing a query. There are three main characteristics of the query federation dataset – content, size and structure – which can be explained as follows:

- **Content**

  With respect to dataset content, data redundancy and distribution can affect the outcome of the source selection. If the same data occurs in multiple sources, then there are more sources that can answer the query. This condition leads to increased communication cost between the federated engine and the SPARQL endpoints. Eventually, the client also receives redundant answers which further increases the communication cost between the client and the federated engine.

  A single query may be able to be answered by multiple sources. Hence, the distribution of data is highly related to the number of SPARQL endpoints that contribute in answering a query. If related data are spread across the dataset, a federated engine should split a query into several sub queries and distribute them to the relevant SPARQL endpoint. As such, the number of requests delivered to the SPARQL endpoints tends to be much higher that actually needed.

- **Size**

  The dataset size of each SPARQL endpoint might be different, for example a SPARQL endpoint handling the smallest dataset may execute a query faster than the others. Inspired by a RDF single repository clustering, we attempt to distribute the dataset size equally amongst the sources to keep the workload balanced amongst the SPARQL endpoints. In terms of the RDF dataset, the dataset size includes the number of triples, number of vertexes, number of properties, number of classes, etc.

- **Structure**

  The structure of an RDF dataset expresses the relationship between properties and entities of the dataset [Duan et al., 2011]. Duan stated that the dataset is more structured, if each entity has all properties that belongs to the same class. To calculate the dataset structure, Duan proposed a metric called *coherence*. In terms of a federated SPARQL query, if the average coherence is close to one, then

the related triples are located in the same source.

- *Q3 : What should be considered when designing a query for federated SPARQL queries benchmark?*

  After addressing the first and second research questions, we formulate a query for benchmarking federated SPARQL queries based on both dataset characteristics and the performance metrics.

  - **Characteristics of the dataset**

    It should be possible to generate queries from any given dataset. In order to address this task, we should have a good knowledge of the characteristics of the dataset beforehand. We can tune the level of complexity of the query based on that knowledge. The characteristics of the dataset used for generating a query are statistical information about dataset content (e.g the frequencies of properties and classes, etc) and distributed properties and classes, etc.

  - **Benchmark metrics** Choosing a specific metric for assessing a system is an individual decision. In some cases, when we consider a metric as an indicator, the result may be compared to another metric result. As an example, if we choose the volume of data transmission between the federated engine and SPARQL endpoints as a metric, then the completeness result might not be achieved when the expected results from the SPARQL endpoint requires more than the capacity of the SPARQL endpoint. Figure 1.4 shows an incoming query asking for information about associated students and courses. The SPARQL endpoint containing student information stores 3000 students, while the SPARQL endpoint about course stores 100 courses that are taken by the students. The federated engine delivers a student query to the Student SPARQL endpoint. The Student SPARQL endpoint only returns 2000 results since the maximum results that can be returned is limited to that number. Consequently, the answer received by the user is incomplete, and to tackle this issue, the federated engine may deliver student query in two steps. However, in this way, the volume of data transmission increases.

    As we described above the completeness results depends on the capacity of SPARQL endpoints. The SPARQL endpoint is generally not in our control, therefore the completeness result is not considered as a metric in this work. Aside

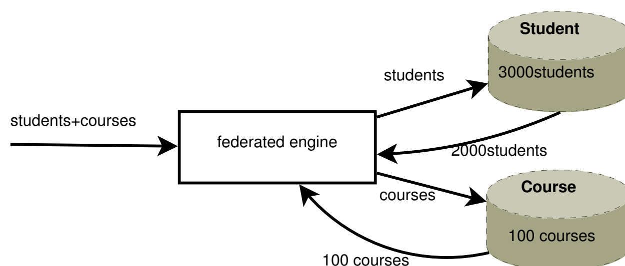


Figure 1.4: Example of query execution for searching students along with their courses

from the completeness results, we do not take into account accuracy as one of main metrics because the data stored in the SPARQL endpoints can change frequently[Umbrich et al., 2010]. Such condition can influence the accuracy of a query results.

The relationship between the metrics has to be carefully investigated if we want to combine them, otherwise, a metric that is more suitable to our situation should be selected. With respect to query generation, we choose the volume of data transmission as a metric because the query federation framework involves network communication in order to execute a query. Based on the first and second research questions, we can investigate which of the performance features influence the volume of data transmission between the federated engine and the SPARQL endpoints.

1.2 Hypothesis

In this thesis, we evaluate several existing federated engines by using various partitioning strategies. The aim of these partitioning strategies is to distribute data across SPARQL endpoints. By distributing data, various of characteristics of datasets, such as the size, content and structure can be generated.

In order to quantify the distribution of the data, a metric is proposed to reveal the characteristics of a dataset. Based on the characteristics of the dataset and selected performance metric, set of queries are generated.

Thus, our hypothesis can be summarised as follows:

> **Given a set of various characteristics of data-sources, we can evaluate a federated SPARQL query framework that take into account the relationship between metrics, queries and datasets.**

1.3 Contribution and Thesis Structure

This thesis aims to present a comprehensive evaluation of a SPARQL query federation framework.

With respect to the *metrics* component, we investigate a set of environment independent metrics, introduce a set of metrics for assessing both the federated engine and the SPARQL endpoints and propose a composite metric for merging different performance metrics.

With respect to the *dataset* component, we design and develop a complementary lightweight tool for generating the dataset benchmark. Furthermore, we propose two spreading factors as dataset metrics for computing the distribution of classes and properties throughout a set of sources.

With respect to the *queries* component, we design a set of queries for benchmarking a specialised SPARQL query federation system which normally has a specific use-case or dataset (e.g., TopFed[22] [Saleem et al., 2014, 2013a], SAFE etc.). As the final output, we provide a tool for generating a set of queries from any given dataset.

To begin with, we carry out a survey to observe existing federated engines. We then compare the works and behaviours of those federated engines (i.e., how they parse queries, select the relevant sources, etc). Finally, we list challenges that should be tackled in the development of the federated engines. This survey is useful to understand the concept of various federated engines. Based on this information, we can design a comprehensive evaluation for those federated engines.

Based on our findings about the general architecture of federated engines in the survey and the data transmission unit, we investigate a set of independent metrics that do not depend on the hardware environment. After that, we propose semi-independent metrics which are derived from the independent metrics. To give a meaningful summary of federated engines comparisons, we then introduce a composite metric. To generate a composite metric, we assign a weight for each performance value of a query and merge the performance metrics values in a single value by using the geometric mean.

Thereafter, we observe the characteristics of a dataset in the federation system environment. Since link between two datasets is one of the characteristics of the dataset, we thus begin by investigating the costs and benefits of link between the datasets while executing a federated query. We then implement nine data distribution strategies to create a dataset for benchmarking. Furthermore, we introduce two spreading factors as dataset metrics, namely

[22]TopFed federation engine: `https://code.google.com/p/topfed/`

the spreading factor of a dataset and the spreading factor of a dataset associated with a query set. The metrics calculation is based on the occurrences of classes and predicates. We also observe the relationship between the spreading factors and communication cost while executing a federated SPARQL query.

We then design a query for benchmarking a SPARQL query federation framework based on the characteristics of the dataset and the communication cost between the federated engine and SPARQL endpoints. To generate a query involving more than one source, we identify the query join patterns from different sources. We then extend the query by adding query features that can increase the communication cost such as the data types and the SPARQL query keywords.

The remainder of the thesis is structured as follows:

- Chapter 2 provides background on Semantic Web standards, in particular, we introduce the RDF and SPARQL technologies. We also introduce the Linked Data concept and the infrastructure for querying over Linked Data. In the last section of this chapter, we focus primarily on the SPARQL query federation approach;

- Chapter 3 presents the state of the art on SPARQL query federation, then we give a comparison between existing federated engines. Additionally, we identify appropriate features and challenges in terms of developing a federated engine;

- Chapter 4 describes existing RDF benchmark suites. Further we provide a list of evaluation approaches to assess the performance of federated engines. Finally, we detail our proposed benchmark metrics;

- Chapter 5 explains the benefits and costs of interlinking in SPARQL query federation. We describe our interlinking definition and investigate the impact of interlinking on the performance of federated SPARQL queries.

- Chapter 6 details several approaches for generating a dataset for benchmarking SPARQL query federation framework.

- Chapter 7 describes our metrics for calculating the distribution of the data in a dataset as a component of the characteristics of a dataset. Finally, we investigate the relationship between the metrics and the performance of a federated engine by executing federated SPARQL queries over the generated datasets introduced in this work.

- Chapter 8 explains a list of requirements for designing a query for benchmarking a SPARQL query federation framework. Later on we detail our methodology for generating queries for benchmarking a federated SPARQL query system;

- Chapter 9 summarises our contributions, discusses the limitations of this work, and suggest future directions.

1.4 Impact

Parts of the work presented herein have been published in various international workshops and conferences, listed here in chronological order.

- we presented our preliminary results that investigate the impact of data distribution on federated SPARQL query at the DIDAS Workshop, IEEE Sixth International Conference on Semantic Computing 2012 [Rakhmawati and Hausenblas, 2012]. This work is presented in Chapter 6;

- we presented a survey of the existing federated engines at the Knowledge Engineering and the Semantic Web Conference 2012 [Rakhmawati and Umbrich, 2013] and then extend it in [Aini Rakhmawati et al., 2013]. This survey is presented in Chapter 3;

- our investigation of metrics for federated SPARQL query benchmark was presented at the Information Systems International Conference 2013 [Rakhmawati, 2013a]. This work is discussed in Chapter 4;

- we presented a study of the costs and benefits of interlinking on federated SPARQL query performance in the International Conference on Web and Information Systems 2014 [Rakhmawati et al., 2014a]. The findings of this study are presented in Chapter 5;

- In the same conference [Rakhmawati et al., 2014b], we introduced a composite metric to merge the performance metric value results into a single metric. This metric is part of the content of Chapter 4. Additionally, we proposed spreading factor metrics for calculating the characteristic of a dataset. Along this line, we provide a tool for generating datasets for federated SPARQL query benchmark. These works are presented in Chapter 6 and 7

- We presented a tool for generating a set of queries for benchmarking a federated engine

in Chapter 8). This tool was presented in the International Conference on Information Integration and Web-based Applications and Services 2014 [Rakhmawati et al., 2014c].

1.5 Thesis Scope

In this study, we focus primarily on federation over SPARQL endpoints infrastructure. Other infrastructures that use query languages such as RQL[23], RDQL[24], SeRQL[25] are beyond the scope of this study. A database benchmark suite is a completed system which consists of test driver, dataset and queries.

With respect to the benchmark, we do not develop a benchmark suite, but we provide a tool for generating a dataset and queries for benchmarking purposes. The generated dataset can not solve the problem in federated Linked Open Data environment (e.g. data distribution) because the dataset generation is only controlled by the Linked Data publisher. The aim of our dataset generation is to have datasets with different characteristics for benchmarking a federated system.

[23]RQL: http://139.91.183.30:9090/RDF/RQL/
[24]RDQL: http://www.w3.org/Submission/RDQL/
[25]SeRQL:http://www.w3.org/2001/sw/wiki/SeRQL

Chapter 2

Background

> *"Knowledge of what is does not open the door directly to what should be."*
>
> —**Albert Einstein**

This chapter describes background contexts of this thesis and introduces several notations that are used in this thesis. In particular, we introduce the Semantic Web concepts: RDF, Turtle and SPARQL. Further, we explain the Linked Data concepts and the infrastructure for querying data over multiple sources.

2.1 Semantic Web

The core idea of the Semantic Web is to make information available on the Web machine-processable [Berners-Lee et al., 2001]. In the early phase, a first solution was to describe web content with metadata which allows a machine to understand the meaning of the content and then process it for other applications.

2.1.1 Resource Description Format(RDF)

In summary, RDF triples are formally defined in Definition 2.1.

Definition 2.1 *Let U be a set of all URIs*[1]*, B be a set of all blank-nodes*[2]*, L be a set of*

[1] URI: `http://www.w3.org/TR/rdf11-concepts/#section-IRIs`

[2] blank-nodes: `http://www.w3.org/TR/rdf11-concepts/#section-blank-nodes`

Figure 2.1: Representation of a RDF statement.

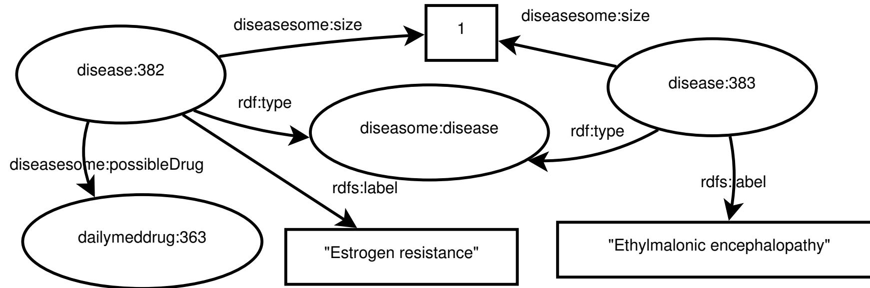


Figure 2.2: Representation of RDF triples

all Literals[3], *then a triple* $tps = (s, p, o) \in (U \cup B) \times U \times (U \cup L \cup B)$ *where* s *is called the subject,* p *is called the predicate and* o *is called the object.*

There are several formats for serialising RDF data, namely RDF/XML [Gandon and Schreiber, 2014], N3 [Berners-Lee, 1998], N-Triples [Beckett, 2014] and Turtle [Beckett et al., 2014]. Throughout this thesis we use Turtle syntax to denote RDF triples, triple patterns and graphs. Given RDF triples as shown in Figure 2.4, then we can write these triples in Turtle syntax as follows:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#label> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix diseasome: <http://wifo5-03.informatik.uni-mannheim.de/diseasome/resource/diseasome> .
@prefix disease: <http://wifo5-03.informatik.uni-mannheim.de/diseasome/resource/diseases> .
@prefix dailymeddrug: <http://wifo5-03.informatik.uni-mannheim.de/dailymed/resource/drugs> .

disease:382 rdf:type diseasome:diseases .
disease:382 rdfs:label "Estrogen resistance"^^xsd:string .
disease:382 diseasome:possibleDrug dailymeddrug:363 .
disease:382 diseasome:size "1"^^xsd:integer .
disease:383 rdf:type diseasome:diseases .
disease:383 rdfs:label "Ethylmalonic encephalopathy"^^xsd:string .
disease:383 diseasome:size "1"^^xsd:integer .
```

In the first five lines, we declare prefixes that are used in RDF triples. The prefixes

[3]literals: http://www.w3.org/TR/rdf11-concepts/#section-Graph-Literal

`rdf`, `rdfs` and `xsd` are frequently used in RDF triples. For the following RDF triples in this thesis, we will omit all prefixes. A full list of prefixes used in this thesis can be found in Appendix A. Literals are declared between double quotes (`""`) as shown in `"Estrogen resistance"` and followed by double-caret delimiter (`^^`) and the data type. We may use either white-space, tabs, or newlines to delimit a subject, a predicate and an object. A dot (`.`) must be appended at the end of each triple.

Further, to abbreviate `rdf:type`, we can replace `rdf:type` with `a`:

```
disease:382 a diseasome:diseases ;
```

Triples which share the same subject can be grouped together using a ; semi-colon delimiter:

```
disease:382 a diseasome:diseases ;
rdfs:label "Estrogen resistance" ;
diseasome:possibleDrug dailymeddrug:363 ;
diseasome:size "1"^^xsd:integer .
```

2.1.2 SPARQL

SPARQL is a query language for RDF. This section presents the syntax and semantics of the SPARQL.

1. **Triple Pattern**

   We initially describe a triple pattern as a core component of a SPARQL query. Like a triple statement, a triple pattern also consists of subject, predicate and object. Apart from those three components, it may contain variables that can be located either on the subject, predicate or object positions, which can be defined as follows:

   Definition 2.2 *A query consists of a set of triple patterns τ which is formally defined as $\tau(s,p,o) \in (U \cup V) \times (U \cup V) \times (U \cup L \cup V)$ where V is a set of all variables.*

   For instance, given the RDF dataset depicted in Figure 2.2, we can retrieve a list of diseases by using the following triple patterns: `?disease` is a variable that can be replaced with any valid value in that RDF dataset. A variable must be started by a question mark(?). Triple pattern `?disease a diseasome:diseases` is matched

Listing 2.1: Example of a SPARQL query for retrieving all triples relating to `diseasome:diseases`

```
select {
?disease a diseasome:diseases .
}
```

against the two following triples: `disease:382 rdf:type diseasome:diseases . disease:383 rdf:type diseasome:diseases .`

2. **Basic Graph Pattern**

   A Basic Graph Pattern (BGP) is a set of triple patterns that is delimited with braces. Like RDF triples, BGP can also be modelled as a graph. The definition of BGP can be found at 2.3. Given the following BGPs in Query 2.2 that aims to retrieve a list of diseases and their drugs from the graph in Figure 2.2, then we can present the above BGP as a graph in Figure 2.3.

   Definition 2.3 *Given a triple pattern* τ *and a Basic Graph Pattern* BGP*,* $var(\tau)$ *is a set of variables in triple* τ *and* $var(BGP) = \cup_{\tau \in BGP} var(\tau)$*. Let* $f(\tau)$ *be a mapping function that generates the triples after replacing* $var(\tau)$ *, then* $f(\tau) = \cup_{\tau \in BGP} f(\tau)$

Listing 2.2: Example of a SPARQL query for retrieving a list of diseases and its drugs

```
SELECT * {
       ?disease a diseasome:diseases .
       ?disease rdfs:label ?name .
       ?disease diseasome:possibleDrug ?drug .}
```

Based on the source graph in Figure 2.2 and the BGP in Figure 2.3, we can see which parts of the source graph are matched against the BGP in Figure 2.4. To begin, we can start matching the first triple pattern of the BGP to the source graph. Then the matching process is followed by the second and third triple patterns of the BGP. Since `disease:383` does not have a property `diseasome:possibleDrug`, we only retrieve the following result: `?disease<-disease:382,?name<-Estrogen resistance,?drug<-dailymeddrug:363`

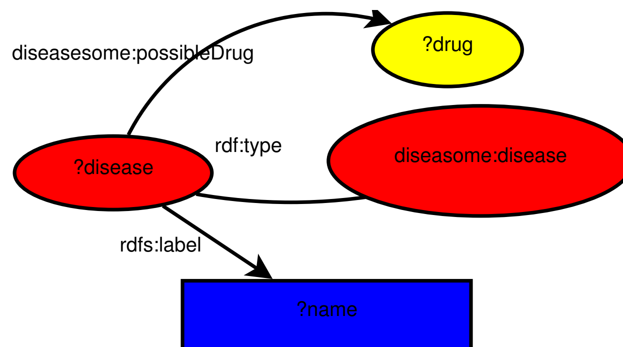


Figure 2.3: Example of a SPARQL basic graph pattern (BGP) represented as a graph.

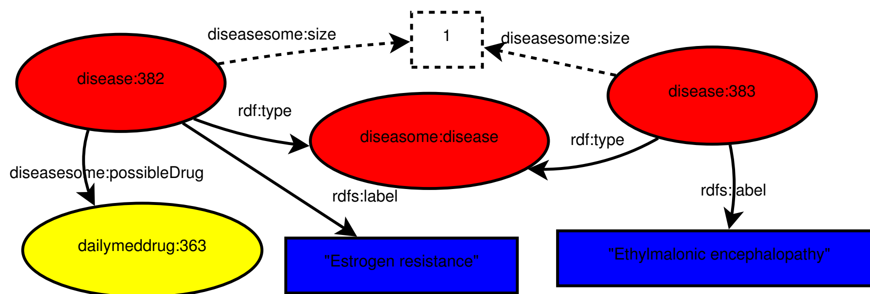


Figure 2.4: Matching parts of the source graph in Figure 2.2 and BGP in Figure 2.3

3. **Joining BGPs**

   This section explains about joining BGPs in a SPARQL query. The joining BGPs can influence performance of a query engine. There are three approaches to join BGPs: *inner join*, *left outer join* and *union*. If we do not specify any operator in-between BGPs, the BGPs are combined by using the inner join scheme. An `OPTIONAL` keyword allows us to do left outer join between BGPs. The `UNION` operator combines the results from two BGPs.

   Looking at our previous query example in Listing 2.2, the results of the query in Listing 2.3 include `disease:383` when we insert an `OPTIONAL` keyword.

   We retrieve the following results from the joining BGPs in Listing 2.3:

   `(?disease<-disease:382,?name<-Estrogen resistance,?drug<-dailymeddrug:363),`
   `(?disease<-disease:383,?name<-Ethylmalonic encephalopathy,?drug<-NULL)`

   Unlike UNION in SQL, we may be able to combine two BGPs with an `UNION` in SPARQL even though the BGPs do not share any common variable. The following BGPs and an `UNION` keyword aim to retrieve results from the source graph in Figure 2.2:

Listing 2.3: Example of a SPARQL query for retrieving a list of diseases and its drugs by using the `OPTIONAL` keyword

```
SELECT *
{
       ?disease a diseasome:diseases .
       ?disease rdfs:label ?name . }
OPTIONAL
{
       ?disease diseasome:possibleDrug ?drug . }
}
```

Listing 2.4: Example of a SPARQL query for retrieving a list of diseases information by using an `UNION` keyword

```
SELECT * {
{ disease:383 rdfs:label ?name . }
UNION
{ disease:382 diseasome:possibleDrug ?drug . }
}
```

Each row of the results has two columns: `?name` and `?drug` :

`(?name<-Estrogen resistance,?drug<-NULL),`

`(?name<-NULL ,?drug<-dailymeddrug:363)`

The previous example might not be a useful query, but it shows that the presence of the `UNION` keywords in the query can return an answer even if both BGPs have different variables.

4. **Query Forms**

   As we stated in Chapter 1, query is one of the main components in a federated SPARQL benchmark system. We will create a query set with various query forms in Chapter 8. In the previous section, we explained the `SELECT` query that is a popular query form in SPARQL that is used to retrieve matched values for variables in the query. SPARQL also supports other query forms, namely:

   - `CONSTRUCT` is similar to `SELECT`, but it returns an answer in an RDF graph. For example, given Figure 2.2 as a source graph, then we ask for all disease triples that have size one by using the query in Listing 2.5. The `CONSTRUCT` query returns

Listing 2.5: Example of a SPARQL CONSTRUCT query for retrieving a list of diseases with size 1

```
CONSTRUCT { ?disease ?p ?o . }
WHERE
{ ?disease diseasome:size 1 .
       ?disease ?p ?o }}
```

seven triples that are exactly the same as the source graph in Figure 2.2.

- ASK aims to check whether or not a given graph pattern matches to any of triples in the source. This query form returns a Boolean value. Suppose that we would like to check the presence of diseases with size one in the source graph in Figure 2.2, then we can deliver the following query to a SPARQL endpoint:

Listing 2.6: Example of a SPARQL ASK query to check a list of diseases with size 1

```
ASK
{ ?disease diseasome:size 1 . }
```

- DESCRIBE retrieves all information about a given resource. For instance, we request all information about disease:383:

Listing 2.7: Example of a SPARQL DESCRIBE query for retrieving all triples relating to disease:383

```
DESCRIBE {  disease:383 }
```

then the SPARQL endpoint returns the following response: disease:383 rdf:type diseasome:diseases .
disease:383 rdfs:label "Ethylmalonic encephalopathy"^^xsd:string .
disease:383 diseasome:size "1"^^xsd:integer .

5. **Retrieving data based on certain condition**

   BGPs matching is inadequate to answer a complex situation. Hence, we can insert a FILTER keyword to retrieve data that meet our specified condition. For example, we execute the following query against the source graph in Figure 2.2:

Listing 2.8: Example of a SPARQL query for retrieving all diseases which disease's size is greater than one by using the `FILTER`

```
SELECT * {
	?disease a diseasome:diseases .
	?disease rdfs:label ?name .
	?disease diseasome:size ?size .
	FILTER (?size > 1)
}
```

The purpose of the query in Listing 2.8 is to retrieve all diseases information which disease size is greater than one. As a result, we obtain an empty result since all size of the diseases in the graph is one.

6. **SPARQL Algebra**

   SPARQL graph pattern can be translated in to a SPARQL algebra form. The SPARQL algebra in this work follows the SPARQL 1.1 document [Harris and Seaborne (eds), 2013] from W3C. A simple BGP pattern (`?s ?p ?o`) can be expressed as *BGP [triple* `?s ?p ?o`*]*. A single BGP may consist of more than one triple. The results of graph pattern matching is grouped by a *ToList* keyword. For instance, query in Listing 2.2 can be translated in Listing 2.9.

Listing 2.9: Example of a SPARQL algebra for query in Listing 2.2

```
(ToList
	(BGP
		[triple ?disease a diseasome:diseases]
		[triple ?disease rdfs:label ?name ]
		[triple ?disease diseasome:possibleDrug ?drug ]
))
```

The *OPTIONAL* graph pattern is left outer join. Thus, the *OPTIONAL* keyword is converted to *leftjoin* keyword in a SPARQL algebra. Aside from that, the OPTIONAL graph pattern is followed by *true* keyword. The example of usage of OPTIONAL SPARQL query in Listing 2.3 can be expressed in a SPARQL algebra in Listing 2.10

In a SPARQL algebra, the *UNION* operator is located before the list of BGPs combined, as seen in Listing 2.4

Listing 2.10: Example of a SPARQL algebra for query in Listing 2.3

```
(tolist
       (leftjoin
              (BGP
              [ triple ?disease a diseasome:diseases]
              [ triple ?disease rdfs:label ?name ]
              )
              (BGP
              [triple ?disease diseasome:possibleDrug ?drug ] )
              true
))
```

Listing 2.11: Example of a SPARQL algebra for query in Listing 2.4

```
(tolist
(union
(BGP
[ triple disease:383 rdfs:label ?name ] )

(BGP
[ triple disease:382 diseasome:possibleDrug ?drug ] )
))
```

The *FILTER* keyword is also written before the set of BGPs that contain the variables in the *FILTER* expression. An example of a SPARQL algebra that contains *FILTER* is shown in Listing 2.12.

Listing 2.12: Example of a SPARQL algebrafor query in Listing 2.8

```
(tolist
       (filter (?size > 1)
              (BGP
              [triple ?disease a diseasome:diseases]
              [triple ?disease rdfs:label ?name ]
              [triple ?disease diseasome:size ?size ]
)))
```

2.2 Linked Data

As the Semantic Web matured, it gave rise to an initiative to make RDF data accessible and browsable. Eventually, such condition will allow us to link between RDF data from various places. This initiative is called Linked Data which is originally inspired by the global network of documents that are connected by hyper-links in the World Wide Web. In Linked Data, structured data in one data source is linked to other structured data from other data sources.

The idea of Linked Data was coined by Tim Berners Lee, who introduced four design rules for publishing the Linked Data [Berners-Lee, 2006]:

1. use URIs as the unique identifier of entities
2. use HTTP URIs so the information about the resource can be looked up on the web
3. use a standard format to publish the information.
4. include other links in the information so that they can discover related information about the resource.

With the advent and rapid adoption of Linked Data, we can now assemble a query over multiple public data sources to retrieve a meaningful and rich information. In the examples above, we only query data from the Diseasome dataset. In order to obtain further information about a drug, we need to combine data from the Diseasome with other datasets such as Dailymed[4] dataset. As illustrated in Figure 2.5, we span the source graph in Figure 2.2. Now we have the RDF data from the Dailymed and the Diseasome datasets. These two datasets are connected by using `diseasome:possibleDrug` predicate.

2.2.1 Querying over Linked Data Infrastructure

As Linked Data grew in popularity, various infrastructures have been designed to cope with data integration between RDF data sources. Based on data source location, the infrastructures for querying Linked Data can be divided in two categories, namely a centralised repositories and distributed repositories.

[4]Dailymed contains information about branded drugs `http://wifo5-03.informatik.uni-mannheim.de/dailymed/`

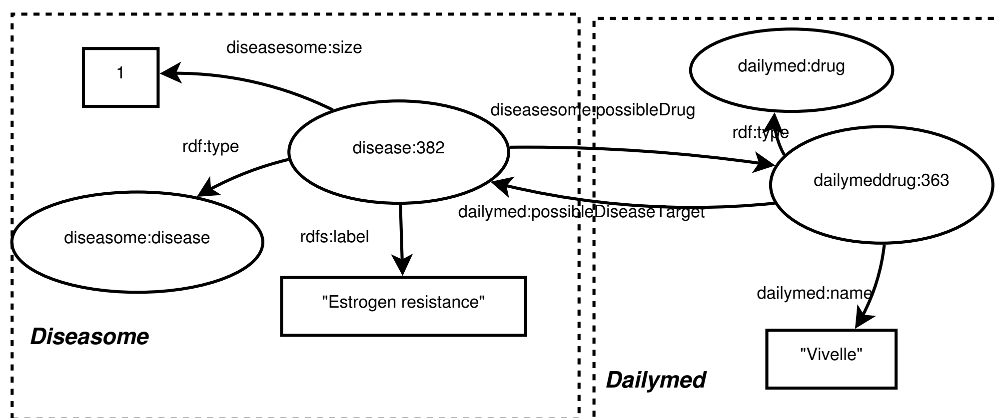


Figure 2.5: Example of Linked Data implementation on Diseasome and Drugbank datasets.

2.2.2 Centralised Repository

As shown in Figure 2.6, the characteristics of a centralised repository are similar to the characteristics of a data warehouse in relational databases, where data are collected in advance before query processing. Watson [d'Aquin et al., 2007] provides a central repository infrastructure for crawling, analysing and indexing RDF Data. Sindice [Tummarello et al., 2007] is another example of a centralised repository which indexes data based on URIs, inverse functional properties, and keywords. Sindice also provides APIs and a SPARQL endpoint for accessing the data.

The single data location used in this infrastructure offers the benefit of not requiring network communication and source selection. As such, the query execution time can be minimised. However, data synchronisation could be a problem in this infrastructure when the data is often changing [Umbrich et al., 2012b].

Another drawback is that this approach is a resource intensive when applied to large scale data. Further, integrating data from multiple sources requires a high maintenance system.

2.2.3 Distributed Repositories

In contrast to a centralised repository, querying over Linked Data in a distributed repositories environment does not require crawling the data beforehand. As sources need not be collected in a single repository, the data is more up-to-date than from a centralised repository's data, but query processing time takes longer. The system only invests little resources such space and time because of no earlier crawling data phase. There are three different systems of distributed repositories: *P2P*, *link traversal* and *federation*, which are discussed in more

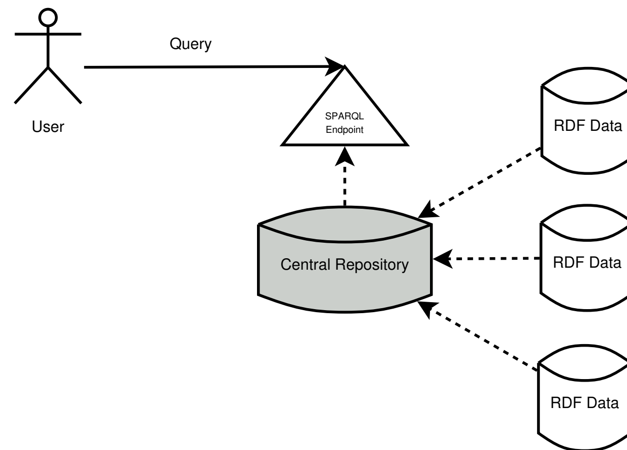


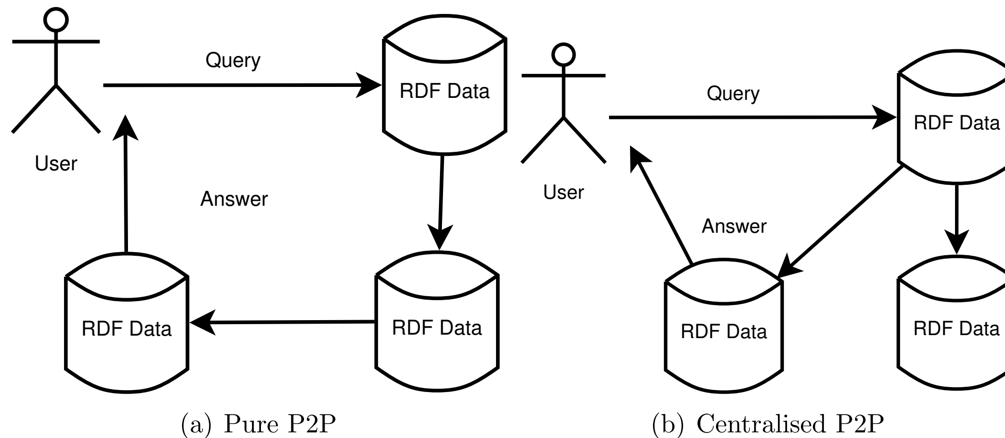
Figure 2.6: Centralised repository

detail in the following subsections.

1. **P2P** P2P is a system that consists of a group of autonomous peers that are connected to each other. These connections allow them to exchange data if they are a pair of peers or they find a route via intermediate peers that can connect them. The P2P can be divided into two types, namely Pure P2P systems and centralised P2P systems. *Pure P2P System* is a system where all peers have the same role without any central component that acts as a mediator in this system. As shown in Figure 2.7(a), all peers communicate with other peers until they find an answer for an incoming query. For centralised P2P systems (Figure 2.7(b)), a peer plays a critical role to index data summary in selecting potential peers that are able to answer a given query. Edutella [Nejdl et al., 2002] implements a centralised P2P paradigm which uses RDF for representing metadata about each peer and exchanging data between the peers.

2. **Link Traversal**

   Discovering data by following HTTP URIs is the basic idea in a link traversal system. As shown in Figure 2.8, without any data knowledge, relevant data sources are detected during runtime execution [Hartig, 2011]. Link traversal provides a high level of freshness of the data since the data is directly accessed from data sources. For executing a query, this system takes a single triple pattern as a starting point. Determining the starting point is a vital task in this system because it influences the flow process of the whole query execution. A wrong starting point can increase the number of intermediate results and eventually increase bandwidth usage. A noteworthy approach



(a) Pure P2P (b) Centralised P2P

Figure 2.7: P2P Systems

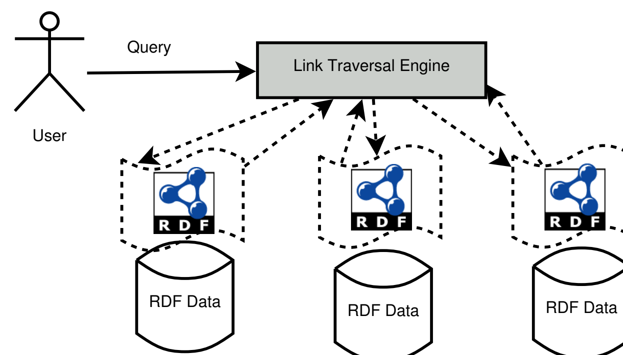


Figure 2.8: Link Traversal

that proposed link traversal is LTBQE [Hartig, 2011]. Later on, Lidaq [Umbrich et al., 2012a] extended LTBQE to support an unbound predicate in query execution.

3. **Federation**

Query federation uses a federated engine as a query mediator to transform a user query into several sub queries and generates results from the integrated data sources. There are two kinds of frameworks for query federation: federation over single repositories (Figure 2.9) and federation over SPARQL endpoints (Figure 2.10). In the federation over single repositories such as Sesame Sail Federation[5], the federated engine delivers sub queries by using a native API. However, not all repositories support this API. A federation over SPARQL endpoints system requires an endpoint as a bridge between

[5]Sesame AliBaba: `http://www.openrdf.org/alibaba.jsp`

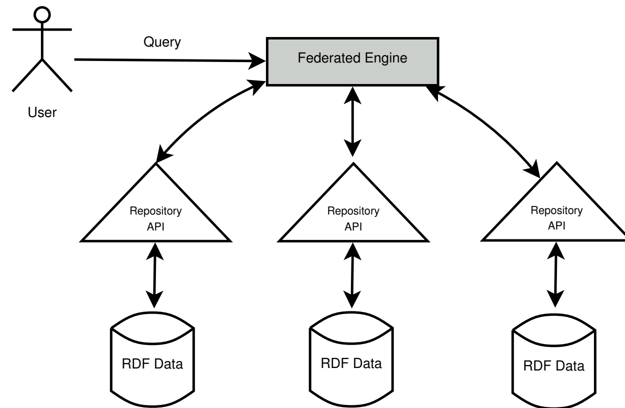


Figure 2.9: Federation over single RDF repositories

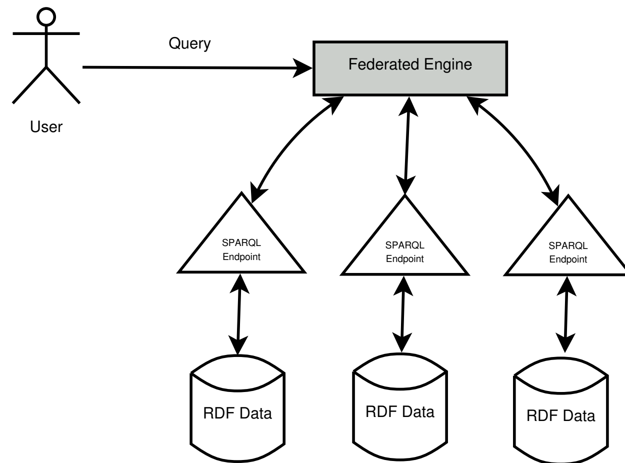


Figure 2.10: Federation over SPARQL endpoints

the federation layer and source (we will detail this type of federation in Chapter 3). In majority, current federated engines are compatible with this type, since RDF stores are generally equipped with a SPARQL Endpoint. Several federated engines [Langegger et al., 2008; Acosta and Vidal, 2011] also offer a wrapper for supporting other data formats like CSV and XML.

Chapter 3

Querying over Federated SPARQL Endpoints —A survey of the State of the Art*

This chapter first discusses federation feature in SPARQL 1.1. We then give an overview of federation architectures, explain six different ways to do query federation, and detail the steps of querying over SPARQL endpoints. Our main contribution of this chapter is a comparison of the different existing federated engines based on their strategies such as source selection and execution plan. By knowing the characteristics of a federated engine, we can identify what factors influence the performance of the federated engine in the rest of this thesis. In the last section, we pointed out the challenges that should be considered in the future development of a federated SPARQL query engine.

3.1 Federation Feature in SPARQL 1.1

SPARQL 1.1 is the currently simplest way to retrieve data from multiple sources. Although SPARQL 1.0 allows us to query data from remote sources, it does not retrieve data from specified remote SPARQL endpoints. It only fetches data from specified remote graphs. Consider the source graph in Figure 2.5, which is extracted from the Diseasome and Dailymed graphs; by using the query in listing 3.1, we can retrieve a list of disease names along with their associated drug names. Notice that, in SPARQL 1.0, we must specify the location of

*Parts of this chapter have been published as [Rakhmawati and Umbrich, 2013; Aini Rakhmawati et al., 2013]

the remote graph after the FROM keyword.

SPARQL 1.1 adds the SERVICE keyword as a way to query data from multiple SPARQL endpoints. However, this requires prior knowledge about the data location, which must be explicitly mentioned in the query. For example, in Listing 3.2, in order to obtain a list of drugs and their associated diseases, the Diseasome and Dailymed SPARQL endpoints are specified after the SERVICE keyword.

Listing 3.1: Example of a federated SPARQL query in SPARQL 1.0

```
SELECT ?diseasename ?drugname {
WHERE {
       FROM <http://localhost/diseasome.rdf> {
       ?disease a diseasome:diseases .
       ?disease rdfs:label ?diseasename .
       ?disease diseasome:possibleDrug ?drug .
}
       FROM <http://localhost/dailymed.rdf> {
       ?drug dailymed:name ?drugname .
}
}
```

Listing 3.2: Example of the same federated SPARQL query in SPARQL 1.1 with the SERVICE keyword

```
SELECT ?diseasename ?drugname {
WHERE {
       SERVICE <http://wifo5-03.informatik.uni-mannheim.de/diseasesome/sparql> {
              ?disease a diseasome:diseases .
              ?disease rdfs:label ?diseasename .
              ?disease diseasome:possibleDrug ?drug .
              }
       SERVICE <http://wifo5-03.informatik.uni-mannheim.de/dailymed/sparql> {
              ?drug dailymed:name ?drugname .
       }
}
```

3.2 Architecture of Federated SPARQL Query Engines

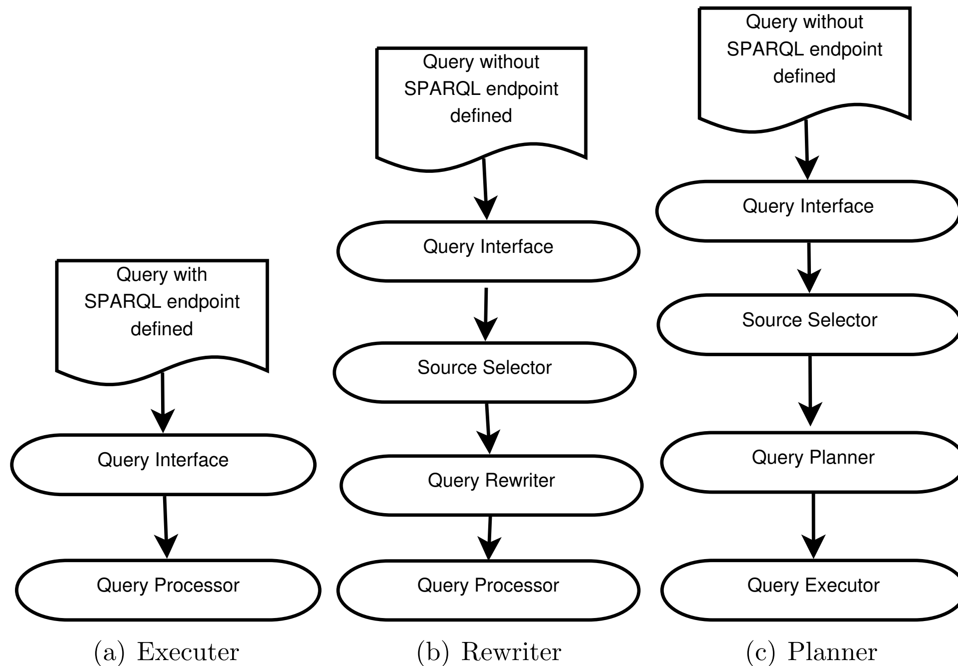
This section describes three types of architecture of federated SPARQL query engines, namely the Executor architecture, the Rewriter architecture and the Planner architecture. SPARQL 1.1 is designed to overcome several limitations of SPARQL 1.0, including update operations, aggregates, and federated query support. We call *Executor architecture* for those systems that already support SPARQL 1.1 because those systems can only execute a query containing `SERVICE` keywords (Figure 3.1(a)) and SPARQL endpoint addresses. A query processor receives a query including the addresses of the SPARQL endpoints, sends each subquery to the defined SPARQL endpoints, and joins the results of the SPARQL Endpoints.

The lack of knowledge about the data being queried is a main problem when executing federated queries. Thus, two architectures have been introduced to overcome this lack of knowledge which we call *Rewriter architecture* (Figure 3.1(b)) and *Planner architecture* (Figure 3.1(c)). A user can write a query blindly without knowing the data location. These federation models can execute a query, such as the one from Listing 3.2 without any SPARQL endpoint declared. By removing the `SERVICE` keyword, the query can be transformed into the query in Listing 3.1. The rewriter architecture provides an interface to translate a query to the SPARQL 1.1 format. After parsing and decomposing the query, the rewriter's source selector determines the most relevant sources for each subquery. The core part of the rewriter architecture is the query rewriter component. This component inserts a `SERVICE keyword` followed by the destination address of each subquery. The result of the query rewriter architecture will be executed by an internal query processor system.

The second type of architecture we described, the planner architecture, creates query execution plans outside of a SPARQL query processor. This architecture was developed while SPARQL 1.1 was still in draft format. The main difference to the rewriter architecture is that the planner architecture must create execution plans, execute each subquery based on the plan and then merge all the results from SPARQL endpoints. All those tasks are done by the query planner and query executor. Consequently, an internal query processor is not required. As in the rewriter architecture, the planner architecture requires a source selector component in order to predict the relevant sources for the query.

Listing 3.3: Example of a federated SPARQL query without the SPARQL endpoint specified

```
SELECT ?diseasename ?drugname {
       ?disease a diseasome:diseases .
       ?disease rdfs:label ?diseasename .
       ?disease diseasome:possibleDrug ?drug .
       ?drug dailymed:name ?drugname .
}
```



(a) Executer (b) Rewriter (c) Planner

Figure 3.1: Architecture of federation over SPARQL endpoints

3.3 Federated SPARQL Query Approaches

Federated SPARQL query may involve more than one SPARQL endpoint. Queries can use several approaches to collect data from multiple SPARQL Endpoints, namely 1) the `SERVICE` keyword, 2) using a variable or popular predicate, 3) the `UNION` keyword, 4) using object similarity and 5) using links. Following some preliminaries, we discuss each of these approaches in turn.

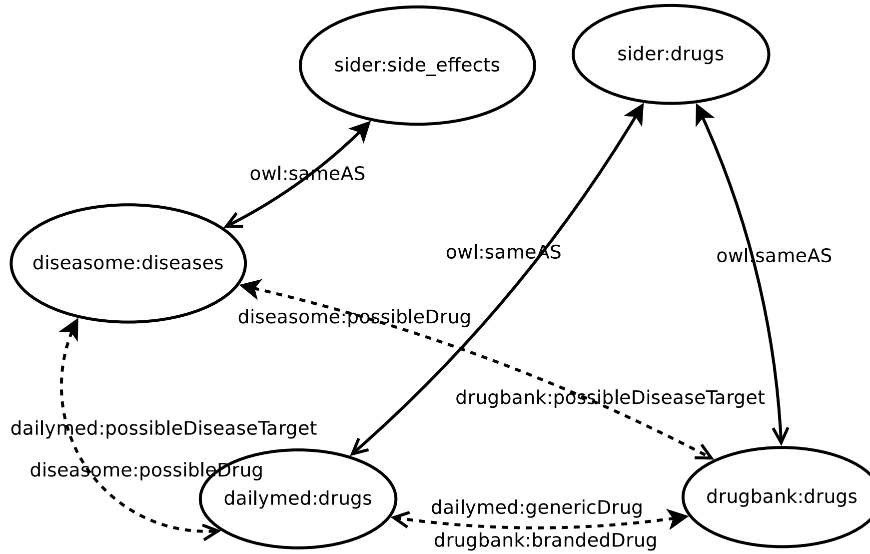


Figure 3.2: The Relationship amongst entities in Drugbank, Disease, Dailymed and Sider datasets

Preliminaries

In order to illustrate these six different approaches to the federation over SPARQL endpoints, we formally define a source-set that is used throughout this thesis as follows:

Definition 3.1 *A source-set D is a finite set of sources d where a source d is identified by a URI. The contents of d (contents(d)) is a set of triple statements t.*

For convenience of notation and if there is no confusion possible, we abbreviate contents(d) with d. For clarity, we formulate queries in this chapter based on the source-set shown in Figure 3.2. Our source-set consists of four life science datasets: Dailymed, Drugbank, Diseasome and Sider[1]. The oval shapes represents the entities, and the arrows represent the interlinking between datasets. Drugbank provides a list of generic drugs information, while Dailymed contains the branded drugs information. Sider consists of a list of side effects of the drugs, whereas Diseasome comprises a list of disorder genes, diseases and the relation between of them.

Except for the `SERVICE` keyword, all other strategies are only able to execute over the federated engine that provides source selection procedure for determining the most relevant source for our query (rewriter engine and planner engine). The `SERVICE` keyword was explained in Section 3.1, we now explain the other five approaches to federated SPARQL

[1]Sider: `http://wifo5-03.informatik.uni-mannheim.de/sider/`

queries: 1) using a variable or popular predicate, 2) the `UNION` keyword, 3) using object similarity and 4) using links.

3.3.1 Using a popular predicate and a variable in the predicate position

Federated engines typically use a list of predicates to determine the relevant sources for a subquery. If the predicate occurs in multiple sources, all of them may be queried. A simple example is shown in Query 3.4 where we can retrieve data from multiple SPARQL endpoints that contain the `rdfs:label` predicate.

Listing 3.4: Example of a federated SPARQL query using the Popular Predicate

```
select * {
       ?thing rdfs:label ?label . }
```

Aside from using a popular predicate, a variable in the predicate position can also combine data from multiple SPARQL endpoints as shown in Query 3.5. The purpose of Query 3.5 is to find any predicate for the drug DB00001 from Drugbank and the drug 1681 from Dailymed dataset that have the same objects.

Listing 3.5: Example of a federated SPARQL query Using a variable in the predicate position

```
select * {
       drugbankdrug:DB00001 ?p1 ?o1 .
       dailymeddrug:1681 ?p2 ?o1 . }
```

3.3.2 Using a `UNION` Operator

The `UNION` operator can combine data from multiple SPARQL endpoints. By using the UNION operator, Query 3.6 obtains the list of diseases for either the drug DB00001 from Drugbank or the drug 1681 from Dailymed.

3.3.3 Object similarity

Determining object similarity by comparing entity labels is another way to aggregate data from multiple SPARQL endpoints. This can be done by a string matching process between

Listing 3.6: Example of a federated SPARQL query using a `UNION` operator

```
select * {
       { drugbankdrug:DB00001 drugbank:possibleDiseaseTarget ?disease .}
       UNION
       { dailymeddrug:1681 dailymed:possibleDiseaseTarget ?disease . }
}
```

two entity labels in different sources. For example, Query 3.7 asks for all Sider drugs that have similar names with Drug DB00316 in the Drugbank dataset. In order to handle case sensitivity, we can use a `filter` operator to compare the objects value. Unfortunately, this strategy requires an expensive query. Furthermore, the entity label is not a unique identifier. Therefore, instead of using the entity name, it is better to use either a literal that contains a unique identifier (such as a serial number) or a URI object such as a homepage address. For instance, Query 3.8 finds all of the branded drugs in the Dailymed dataset corresponding to all of the diseases in the Diseasome dataset by mapping the object of `diseasome:possibleDrug` to the object of `dailymed:genericDrug`.

Listing 3.7: Example of a federated SPARQL query using label comparison

```
select * {
       drugbankdrug:DB00316 rdfs:label ?drugname .
       ?siderdrug a sider:drugs .
       ?siderdrug rdfs:label ?drugname . }
```

Listing 3.8: Example of a federated SPARQL query using URIs comparison

```
select * {
       ?disease a diseasome:diseases .
       ?disease diseasome:possibleDrug ?genericdrug .
       ?drug a dailymed:drugs .
       ?drug dailymed:genericDrug ?genericdrug . }
```

3.3.4 Using Links

The aims of *links* generation are 1) to connect two entities having the same characteristic(e.g `owl:sameAs`), 2) to link two related entities (e.g. `drugbank:possibleDiseaseTarget`).

Hence, the existence of *links* can be useful to merge data from multiple SPARQL endpoints. Like Query 3.7, Query 3.9 also retrieves drugs that are the same as drug DB00316 in the Drugbank by using the `owl:sameAs` *link*.

Listing 3.9: Example of a federated SPARQL query using links

```
select * {
       drugbankdrug:DB00316 owl:sameAs ?siderdrug .
       ?siderdrug a sider:drugs .
       ?siderdrug rdfs:label ?drugname . }
```

3.4 Basic Steps of Query Process in a Federated Engine

We have introduced five different approaches to query in a federated SPARQL query framework, we next introduce the basic steps of federated querying. As the mediator, the federated engine holds three important roles: 1) to manage an incoming query from a user, 2) deliver the query to each source and 3) give back the results to the user. We explain some important concepts about query federation frameworks for executing a query, as depicted in Figure 3.3, the steps in federated querying are 1) query parsing, 2) source selection, 3) source tracking 4) query planning and 5) query execution; we describe each of these steps in turn.

3.4.1 Query Parsing

In this initial phase, SPARQL query is transformed to an internal pattern, which can be in the list of BGPs (Section 2), abstract syntax tree [Görlitz and Staab, 2011], or other formats. The outcome of the parsing step is useful for later steps, particularly in query optimisation.

3.4.2 Source Selection

Instead of sending every piece of a query to all sources, a federated engine should determine the relevant source for each subquery carefully. It may be acceptable to deliver a simple query to all destinations, however, a complex query with many intermediate results could lead to increased communication cost due to more transmissions between the federated engine and SPARQL endpoints. Further, the capacity of each source to answer the query should also be considered. Choosing a relevant source for a query can be done in four different ways: ASK query, data catalogue, data indexing, and caching.

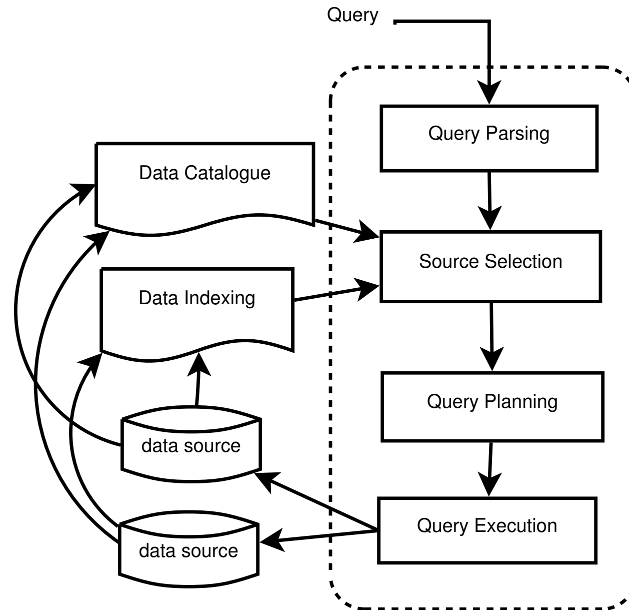


Figure 3.3: Federated SPARQL query Process in a federated engine

- ASK Query
  An ASK SPARQL query returns a Boolean value that indicates whether or not a query can be answered by the SPARQL endpoint. For instance, to solve the query in Listing 3.3, a federated engine will send a query in Listing 3.10 to the SPARQL endpoints in order to seek sources that can answer the subquery `?drug dailymed:name ?drugname`. By sending an ASK query, the bandwidth usage can be reduced significantly because a subquery is only transferred to the endpoints responding *true*. Furthermore, sending an ASK query can detect the changing of data at runtime. The limitation of SPARQL ASK is that it is only a binary decision and it cannot detect redundancy data among sources. To address redundancy, [Hose and Schenkel, 2012] extended the ASK operation during source selection by including a sketch: an estimation of the number of results and a summary of the results. Another limitation of ASK query is that it is only suitable for SPARQL query federation frameworks with a small number of sources because it takes a long time to wait for each SPARQL Endpoint to answer the ASK query.

- Data catalogue
  By consulting the source catalogue, a federated engine can predict suitable sources for a query. For example, [Langegger et al., 2008; Görlitz and Staab, 2011] use VoID [Alexander and Hausenblas, 2009] as the data catalogue. VoID expresses the metadata of the

Listing 3.10: Example of an ASK query to select the most relevant source

```
ASK {
      ?drug dailymed:name ?drugname .
}
```

dataset and its relation with other datasets, according to LOD cloud statistics, 32.2% of LOD datasets provide a dataset description expressed in VoID. The purpose of VoID is used to register a new source in the SemWIQ [Langegger et al., 2008], and to estimate a cardinality for a SPARQL query pattern [Neumann and Moerkotte, 2011] in SPLENDID [Görlitz and Staab, 2011]. SPLENDID and SemWIQ will be further explained in Section 3.5. Another way to provide information about a dataset is to use RDFStats [Langegger and Woss, 2009] which is built on SCOVO [Hausenblas et al., 2009]. RDFStats consists of a list of statistics of instances and a histogram of classes, properties and value types. Another data catalogue is Service Description [2] which describes the data availability from a SPARQL endpoint, data statistic and the restriction of query pattern; it is primarily designed for [Quilitz and Leser, 2008]. Most data catalogues only provide a list of predicates since the number of predicates is less than the number of subjects and objects. Based on the catalogue, the federated engine generally pre-compute the statistics that are needed for query optimisation. This data catalogue can be updated during the query execution, especially for frequently changing data. The freshness of a data catalogue impacts the accuracy of source selection, but the updating process consumes significant bandwidth. Furthermore, generally, the SPARQL query delivered for updating data catalogue is an expensive operation which might be refused by the SPARQL endpoint.

- Data Indexing
  According to the LOD cloud statistics, 63% of sources do not expose their data catalogues. In order to overcome the lack of data catalogues, data indexing could occur either before or during query execution. [Harth et al., 2010] proposed an indexing method that applies QTrees [Hose et al., 2007]. Indexing could assist a federated engine to determine the relevant sources for a query by describing the instances and schemas contained in each dataset. The data is normally indexed periodically. Thus, the advantage of data indexing is data freshness. In general, the drawback of data

[2]Service Description: http://darq.sourceforge.net/#Service_Descriptions

indexing is that it needs more storage compare to data catalogue as it usually indexes every triple. In order to deal with this storage problem, [Basca and Bernstein, 2010] detects the relevant query sources by obtaining dataset descriptions from either a Web Directory or a Search Engine such as Sindice[3] after the query parsing step. These indexes provide a summary of ontological prefixes, predicates and classes which can be used for the query execution process. However, network latency issues arises during execution because this approach relies on third-party indexing.

- Caching
  Most federated engines load statistical information during the initialisation phase because it may reduce the bandwidth consumption at runtime. However, the source selection process is not accurate, if the data is often changing. To tackle this problem, the caching can be used during query execution to improve the freshness of the statistics used in later query execution. The information stored in a cache is likely to be similar to that stored in a data catalogue. To decrease the communication cost, a federated engine does not deliver a new query, but rather updates statistical source information based on the results of subqueries answers from the SPARQL endpoints.

All the aforementioned source selection strategies can be applied to dynamic data such as data streams, as long as either data catalogue or indexing is always updated. Because updating tasks consume significant bandwidth, we should have a mechanism that only requests the frequently changing data as proposed by [Umbrich et al., 2012b] or that only updates data information periodically.

3.4.3 Source Tracking

Source tracking is an approach to detect the availability of SPARQL endpoints. In order to determine whether a SPARQL endpoint is alive or not, a federated engine can send SPARQL ASK queries periodically. Besides, the SPARQL endpoint can use a cache for storing information about previous query execution. If a SPARQL endpoint can answer in the previous query execution, the SPARQL endpoint can send the current queries.

[3] Sindice: `http://sindice.com/`

3.4.4 Query Planning

In the second phase of query processing, a federated engine decomposes the query and builds multiple sub queries based on the results of the source selection step. The construction of subqueries is also considered before the query is transmitted. A single triple pattern can match either a single source or multiple sources. To reduce redundancy sending a single query pattern to multiple sources, a query pattern can be grouped with other query patterns in a single subquery. Grouping query patterns can also minimise the selectivity value of query and ultimately reduce the intermediate join processes. To cluster related query patterns in a single subquery[Schwarte et al., 2011] and [Akar et al., 2012] proposed the *Exclusive Group scheme*. The triple patterns that can be answered by a single source and have the same variable are grouped in a single subquery.

In order to arrange the order of a list of join arguments, a federated engine builds several query execution plans that are generally represented by a tree structure. The subquery combination produces various query execution plans. Hence, a federation engine normally uses statistical information to compute the cost execution of each plan. The cost of the execution can be calculated, for example by cardinality estimation, selectivity estimation, etc. This process influences the number of intermediate results produced at runtime. The objective of this step is to select the best execution plan with lowest execution cost (fewest number of intermediate results).

3.4.5 Query Execution

Herein, we describe several strategies for delivering subqueries to SPARQL endpoints:

- Nested Loop Join
  Nested loop join is not an optimal solution for a complex query since each previous scanning result will be joined to the next results.

- Bind join
  Improving on the nested loop join, the bind join, first introduced by [Haas et al., 1997], uses the intermediate results as a filter for the subsequent queries. As a result, the transmission cost can be minimised, but the query runtime increases because the query mediator has to wait for the complete answer of each previous query.

- Bound join
  A variation of the bind join strategy, the bound join [Schwarte et al., 2011] combines

the intermediate sub queries to a single SPARQL endpoint into a single subquery by using the `UNION` keyword.

- Hash join
  Hash join, implemented in [Görlitz and Staab, 2011], submits sub queries in parallel and then joins all intermediate results locally. This join can boost the runtime performance for the small intermediate results. However, the transmission cost will be higher if the intermediate results are large.

3.5 The Existing Frameworks for Federation over SPARQL Endpoints

As described in the Section 3.2, there are three federation architectures that have been developed recently, namely executor architecture, rewriter architecture and planner architecture. In this section, we only explain the existing federated engines that are classified as rewriter architecture and planner architecture. These federated engines are compared in Table 3.1 and 3.2 based on seven dimensions: data catalogue, source selection strategy, platform, the presence of cache, query execution strategy, source tracking and whether they have a Graphical User Interface (GUI). We do not detail the executor architecture because most SPARQL query processors such as ARQ[4], Sesame[5], Virtuoso[6] are able to execute a query with the `SERVICE` keyword.

3.5.1 Rewriter architectures

Several frameworks have been built on top of SPARQL query engines supporting SPARQL 1.1 such as ARQ, Sesame and Virtuoso. In order to execute a federated SPARQL query that does not specify endpoints, these engines rewrite the query by appending the `SERVICE` keyword. Table 3.1 compares federated engines that implement the rewriter architecture approach. First, we look at the data catalogue used by the engines. Then we compare the platforms that are used to develop the engine. The source selection column describes how the engine determines the relevant sources. We then look at whether a cache is present during query execution. The query execution plan strategies are also compared. The source

[4]ARQ: `http://jena.apache.org/documentation/query/index.html`
[5]Sesame: `http://www.openrdf.org/index.jsp`
[6]Virtuoso: `http://www.openlinksw.com/`

Engine	Catalogue	Platform	Source Selection	Cache	Query Execution	Source Tracking	GUI
SemWIQ	RdfStats + VoID	Jena	Statistic + Service	✓	Bind Join	✓	✓
Anapsid	Predicate List and Endpoint status	Anapsid	Predicate List	X	Symmetric Hash Join and XJoin	✓	✓
WoDQA	VoID Stores	Jena	List of predicates and ontologies	X	X	✓	✓

Table 3.1: The existing federated engines using the rewriter architecture approach.

selection column has two values: dynamic and static. The dynamic value means that the changing data in sources or SPARQL endpoints are updated automatically, while the static value means that information about the sources is never updated after the federated engine is executed. The last column indicates whether a GUI is available. Table 3.2 also contains these columns.

ANAPSID

The ANAPSID framework [Acosta and Vidal, 2011] is designed to manage query execution with respect to the data availability and runtime conditions for the SPARQL 1.1 federation. It enhances the XJoin [Urhan and Franklin, 2000] operator and combines it with Symmetric Hash Join [Deshpande et al., 2007]. Both of them are non-blocking operators that save the retrieved results to a hash table. Similar to other frameworks, it also has a data catalogue that contains a list of predicates. Additionally, execution time-out information of the SPARQL endpoint is added to the data catalogue. Therefore, the data catalogue is updated on the fly. Apart from updating a data catalogue, ANAPSID also updates the execution plans at runtime. The Defender [Montoya et al., 2012a,b] in ANAPSID translates an incoming query to SPARQL 1.1 format. Further, the Defender also composes related subqueries in the same group using bushy tree algorithm [Vidal et al., 2010].

SemWIQ

SemWIQ [Langegger et al., 2008] is another system building on top of ARQ and part of the Grid-enabled Semantic Data Access Middleware (G-SDAM). It provides a specific wrapper to allow the source without an equipped SPARQL endpoint connected.

Query federation relies on data summaries in RDFStats and SDV[7]. RDFStats always updates the statistical information since the SemWIQ monitoring component periodically collects information at runtime and stores it into a cache. As the RDFstats also covers histograms of String, Blank Nodes etc, it is more beneficial for SemWIQ to be able to execute any kind of query pattern. SDV is based on VoID which is useful for source registration. A query is parsed by a Jena SPARQL processor ARQ before the optimisation process. SemWIQ applies several query optimisation methods based on statistical cost estimation such as push-down of filter expressions, push down of optional group patterns, push-down of joins and join and union reordering. During optimisation, the federator component inserts the `SERVICE` keyword and a SPARQL endpoint address for each subquery.

WoDQA

WoDQA (Web of Data Query Analyzer) [Akar et al., 2012] also uses ARQ as the query processor. The source selection is done by analysing the metadata in the VoID stores such as CKAN[8] and VoIDStore[9]. Source observation is based on Internationalised Resource Identifiers (IRI), linking predicates and shared variables. It does not exploit any statistical information from the VoID of each source, but only compares either IRIs or linking predicates to subject, predicate and object. The same variables in the same position are grouped in a single subquery. After detecting relevant sources for each subquery, the SERVICE keyword is appended followed by the SPARQL endpoint address.

3.5.2 Planner Architectures

In this architecture, a federated engine acts as a mediator [Hose et al., 2011] that transfers a SPARQL query from the user to multiple sources either single RDF repositories or SPARQL endpoints. Before delivering a query to the destination source, a planner architecture breaks down a query into sub queries and selects the destination of each subquery. Following query execution, it must join the results retrieved from the SPARQL endpoints. The following overview of the current federated engines that implement the planner architecture is summarised in Table 3.2.

[7]SDV: `http://purl.org/semwiq/mediator/sdv#`
[8]CKAN: `http://ckan.net/`
[9]VoIDStore: `http://void.rkbexplorer.com/`

DARQ

Distributed ARQ (DARQ) [Quilitz and Leser, 2008] is an extension of ARQ which provides a transparent query access to multiple distributed endpoints. A Service Description which consists of the data description and statistical information has to declare in advance before query processing to decide where a subquery should go. According to the list of predicates in the Service Description, it re-writes the query, creates sub queries and designs the query planning execution. The query planning is based on estimated cardinality cost. DARQ implements two join strategies: Nested Loop Join and Bind Join (Section 3.4.5).

Splendid

Splendid [Görlitz and Staab, 2011] extends Sesame which uses VoID as the data catalogue. The VoID of source is loaded when the federated system is started and ASK queries are submitted to each source for a verification. Once the query is issued, the system builds sub queries and determines the optimal join order. Based on the statistical information, the bushy tree execution plan is generated by using dynamic programming [Selinger et al., 1979]. Similar to DARQ, it computes the join cost based on cardinality estimation. It provides two join types to merge the results locally, namely hash join and bind join.

FedX

FedX [Schwarte et al., 2011] is also developed on top of the Sesame framework. It is able to run queries over either Sesame repositories or SPARQL endpoints. During the initial phase, it loads the list of sources without their statistical information. The source selection is done by sending SPARQL ASK queries. The result of a SPARQL ASK query is stored in a cache to reduce communication for the subsequent queries. A rule based join optimiser minimises intermediate result size based on the cost estimation. FedX implements exclusive groups to cluster related patterns for a single relevant source. Beside grouping patterns, it implements a bound join strategy. Those strategies can decrease the number of query transmission and eventually, they reduce the size of intermediate results. It comes with a workbench as the GUI for demonstrating FedX's federated approach to query processing.

ADERIS

Adaptive Distributed Endpoint RDF Integration System (ADERIS) [Kikuchi et al., 2010] fetches the list of predicates provided by a source during the setup stage. The

predicates list can be used to determine the destination source for each subquery pattern. During query execution, it constructs predicate tables to be added in the query plan. One predicate table belongs to a subquery pattern. The predicate table consists of two columns: subject and object, which are filled from the intermediate results. Once two predicate tables have been completed, the local joining starts using a nested loop join algorithm. The predicate tables are deleted after the query is processed. ADERIS is suitable for a source that does not expose any data catalogue, but it only handles limited query patterns such as UNION and OPTIONAL. A simple GUI for configuration and query execution is provided.

Avalanche

Avalanche [Basca and Bernstein, 2010] does not maintain a source registry because the information about sources comes from third parties such as search engines and web directories. Beside these, Avalanche also stores a set of prefixes and schemas for special endpoints. The statistics about sources are always up to date because Avalanche always requests source information from the search engine or the web directory after query parsing. To detect the sources that can contribute to answering a subquery, it calculates the cardinality of each unbound variable. The combinations of sub queries are constructed using best first search approach. All subqueries are executed in parallel. To reduce the query response time, Avalanche only retrieves the first K results.

GDS

Graph Distributed SPARQL (GDS) [Wang et al., 2011a] overcomes the limitation of their previous work [Wang et al., 2011b] which does not handle multiple graphs. It is developed on top of the Jena platform by implementing the Minimum Spanning Tree (MST) algorithm and enhancing a BGP representation. Based on the Service description, a MST graph is generated by exploiting the Kruskal algorithm which aims to estimate the minimum set of triple patterns evaluation and the best execution order. The query planning execution can be done by either semi join or bind join which is assisted by a cache to reduce the traffic cost.

Sesame

As early as 2009, Sesame supported federated SPARQL querying by using SAIL AliBaba extension[10], but it could not execute a query containing the `SERVICE` keyword.

[10]Sesame SAIL Alibaba: `http://www.openrdf.org/doc/alibaba/2.0-alpha2/alibaba-sail-federation/index.html`

Engine	Catalogue	Platform	Source Selection	Cache	Query Execution	Source Tracking	GUI
DARQ	Service Description	Jena	Statistic of Predicate	✓	Bind Join or Nested Loop Join	X	X
ADERIS	Predicate List during setup phase	X	Predicate List	X	Nested Loop Join	X	✓
FedX	X	Sesame	ASK	✓	Bind Join parallelisation	✓	✓
Splendid	VoID	Sesame	Statistic + ASK	X	Bind Join or Hash Join	X	X
GDS	Service Description	Jena	Statistic of Predicate	✓	Bind Join or Semi Join	✓	X
Avalanche	Search Engine	Avalanche	Statistic of predicates and ontologies	✓	Bind join	✓	X
Sesame AliBaba	X	Sesame	X	X	Nested Loop Join	X	X

Table 3.2: The existing federated engine applying the planner architecture approach.

Instead, for SPARQL 1.0, Sesame integrates multiple sources into a virtual single repository to execute a federated query. Its API can execute federated SPARQL queries either over RDF dumps or SPARQL endpoints. The source must be registered in advance during a setup phase. By default, Sesame provides a simple configuration file containing only a list of SPARQL Endpoint addresses. Thus, Sesame sends queries to all sources without a source selection mechanism. In order to optimise the query execution, Sesame offers additional features in the configuration file, namely a list of predicate and subject prefixes, which is used to predict the relevant source for a subquery based on prefix matching. The join ordering is decided by calculating the size of basic graph patterns. The latest version of Sesame provides a federated query feature.

3.6 Challenges

Although federation over SPARQL endpoints has been actively developed over the past few years, particularly to design source selection algorithms, this area is still in its infancy. In future development, several challenges need to be tackled, which were highlighted by the survey in Section 3.5.

- Data catalogues
  As described in Section 3.5, source registration can be done by a federated engine as well as by third parties such as search engines. For querying over Linked Open Data, the source registry should not be limited. Federated engines could combine both static and dynamic source registration where sources in the static registration are given a higher priority than source in the dynamic registration. Assigning different priority levels is to ensure the federated engine to get the potential sources for a sub-query. Updating a data catalogue requires an expensive query. Hence, there should be mechanisms to reduce the communication cost between a federated engine and SPARQL endpoints while the federated engine is updating the data catalogue.

- Data Access and Security
  The sources and federated engine are usually located in different locations, therefore secure communication processes should be used for sending data between the federated engine and sources. Several SPARQL endpoints restrict query access to limited users based on authentication features. However, to the best of our knowledge, no federated engine protects against an unauthorised interception between the federated engine and sources. Public key cryptography could be implemented in the query federation frameworks where the federated engine and sources share public and private keys for data encryption during interaction.

- Data Allocation
  Since several RDF stores crawl data from other sources, data redundancy can not be avoided in the Linked Open Data Cloud. Consequently, a federated engine may detect the same data from multiple locations. This could increase the communication cost during selection source and query execution stage, particularly for federated engines that use a statistical information from a third party. Furthermore, this redundant data could increase the intermediate results. On the one hand, using a popular vocabulary allows user to query heterogeneous sources [Polleres, 2010], but on the other hand,

source prediction for a query is a hard task. As pointed out by [Rakhmawati and Hausenblas, 2012] when popular entities and vocabularies are distributed over multiple sources, the performance of federated queries decreases.

- Data Freshness
  Freshness is one of the most important measures for the data integration because each source might have different freshness values. Having an up-to-date data catalogue is essential for a query federation framework to achieve a high freshness value. Inaccurate results could arise from an inaccurate data catalogue. Nevertheless, updating a data catalogue is a costly operation in terms of the query execution and traffic between the sources and federated engine. Apart from data catalogue problem, a high level of freshness cannot be obtained when high network latency occurs during the communication process.

- Benchmarks
  To date, FedBench [Schmidt et al., 2011a] is the only benchmark proposed for evaluating federated query performance. It provides two performance metrics: loading time and querying time. Those performance metrics are not sufficient to evaluate a query federation framework. Several performance measurements from traditional distributed database should be considered such as query throughput. In addition, several metrics that are related to the approach that is implemented in each phase in a federated engine should be considered, such as the size of intermediate results, number of requests, amount of data sent, etc. Apart from the performance metrics, data quality metrics become important —due to data heterogeneity; data quality metrics include freshness, consistency, completeness and accuracy. FedBench metrics were updated by [Montoya et al., 2012a]— who added two more FedBench metrics, namely Endpoint Selection time and completeness. Furthermore, it evaluated the performance of the federated engines in various environments. Since the FedBench has a static source-set and query set, it is difficult to evaluate a federated engine by using another source-set. To address this problem, the SPARQL Linked Open Data Query Generator (SPLODGE) [Görlitz et al., 2012] generates a set of random queries for any given dataset. The query set generation is based on the characteristics of a dataset that are obtained from its predicate statistics. Besides the characteristics of the dataset, SPLENDID also considers the query structure and complexity (i.e the number of joins, query shape, etc) to produce the query set. Using similar idea, we propose QFed (Chapter 8), a lightweight tool for

generating queries benchmark based on communication cost and characteristics of a dataset.

- Overlapping Terminologies
  The RDF data are generated, presented and published using numerous expressions, namespaces, vocabularies and terminologies, that significantly contain duplicate or complementary data [Quackenbush, 2006; Bechhofer et al., 2011]. Therefore, the mapping rules among heterogeneous schemas is highly required in query federation. This task could be done by having a global schema catalogue —that maps related concepts or properties— and generating more links among the related entities.

- Provenance
  Since more than one source is involved in a federated SPARQL query, the origin of data results is not prominent. Further, data redundancy often results from a federated SPARQL query, especially in federation over Linked Open Data. This is because several publishers expose the same dataset. For example, Sindice contains DBpedia data: when a user requests DBpedia data, both DBpedia and Sindice SPARQL endpoints are able to answer that query. Such condition can cause the redundant results which cannot be avoided by the query federation frameworks that use third party catalogues. Hence, the data provenance is an important factor in the federation over SPARQL endpoints. A notable provenance implementation in the OPENPHACTS[11] federation system is explained in [Harland, 2012]. In order to tackle the provenance issue, OPENPHACTS uses the Nanopublication [Groth et al., 2010] format which supports provenance, annotation, attribution and citation.

3.7 Conclusion

Federated SPARQL queries have made significant progress in recent years. Although a number of federated engines have already been developed, the field is still relatively far from its maturity. Based on our experience with the existing federated engines, most federated engines focus mostly on source selection and join optimisation during query execution.

In this chapter, we listed federated engine and their features. We classified those frameworks into three categories, namely executor architecture, rewriter architecture, and planner architecture. Based on this list, a user can choose a suitable federation engine for their need.

[11]OPENPHACTS: http://www.openphacts.org/

Based on the current generation of query federation frameworks surveyed in this work, further improvements are still required to make the frameworks more effectively in a broader range of applications.

Based on the survey in this chapter, we then determine a suitable metric for assessing performance of the federation over SPARQL endpoints framework in Chapter 4. Furthermore, designing a benchmarking query for federated SPARQL query (Chapter 8 also requires a comprehensive understanding of the work of a federated engine that we have described in general architecture of the federated engine.

Chapter 4

A Proposal for New Performance Metrics for Query Federation Benchmarks*

In this chapter we first give an overview of the state of the art in RDF benchmarks for both a single repository and distributed repositories. We then list the existing evaluations of query federation frameworks. We classify a set of basic performance metrics based on their measurement regions. Later on, we discuss what metrics should be considered in assessing a framework for federation over SPARQL endpoints, namely independent metrics, semi-independent metrics and composite metrics. We conclude this chapter by providing an evaluation of existing federated engines by using those three metrics.

4.1 State of the art of RDF repository benchmarks

With the popularity of Semantic Web, the need for RDF store benchmarks has been acknowledged. As there are some similarities between the Semantic Web and relational databases, some current RDF benchmarks adopt relational database benchmarks such as performance metrics, sets of queries, synthetic datasets etc. The Benchmark Handbook [Gray, 1992] explains a list of notable benchmarks for relational database such as TPC, Wisconsin [DeWitt, 1993], SPEC [Dixit, 1991].

In this section, we conduct a survey on the state of the art in RDF repository benchmarks. In particular, we explain the metrics, queries and datasets used in current benchmarks.

*Parts of this chapter have been published as [Rakhmawati, 2013a; Rakhmawati et al., 2014b]

4.1.1 Benchmark for a single RDF repository

Although our concern is federation over SPARQL endpoints, we also describe benchmarks for single RDF repository because they were used to evaluate some initial work in query federation.

1. **Lehigh University Benchmark (LUBM)** LUBM [Guo et al., 2005] provides a synthetic dataset that contains university data and activities. This dataset is parametrisable which allows us to customise the dataset. It provides 14 test queries, but those queries do not include `OPTIONAL`, `UNION`, and solution modifiers. To test performance of a RDF repository, they proposed six metrics, as follows:

   - Load Time: How long it takes to load dataset into the repository in initial setup.
   - Repository Size: How large the secondary storage is after loading data
   - Query Response Time: How long it takes for the federated engine to return the final results after an incoming query received from the user.
   - Query Completeness: The percentage of the entailed answers that are returned by the system
   - Query Soundness: The percentage of the answers returned by the system that are actually entailed.
   - Combined metric: To give a meaningful summary of all performance metrics, LUBM combines query response time, completeness and soundness into a single metric by using *F-measure* [van Rijsbergen, 1979] formula.

2. **Berlin SPARQL Benchmark (BSBM)** The Berlin SPARQL Benchmark (BSBM) [Bizer and Schultz, 2009] consists of 25 queries that are grouped into 12 query types. The queries demonstrate the use case of e-commerce search. The BSBM dataset contains a set of products that is offered by different companies and a set of product reviews from consumers. Furthermore, it also provides a driver test —for setting up a test environment, performing several tests and concluding the final performance results. Like LUBM, the loading time is also used as a metric in BSBM. Additionally, BSBM measures how many queries can be executed within one hour.

3. **SPARQL Performance Benchmark (SP2B)** SPARQL Performance Benchmark (SP2B) [Schmidt et al., 2008] is intended for benchmarking a single RDF repository that supports SPARQL querying. Thus, the 14 test queries provided cover all the

key operations in SPARQL 1.0. Similar to LUBM and BSBM, it provides a synthetic dataset, but the SP2B dataset reflects real world cases since the dataset is originally from the DBLP dataset[1]. Various sizes of the dataset can be generated. More comprehensive metrics are proposed by SP2B to assess the performance of a single RDF repository:

- Success Rate: This metric illustrates the success rate of a single query execution. It has four different values: success, time-out, memory exhaustion and error.
- Loading Time
- Per Query Performance: This represents the performance of an RDF repository in executing a single query.
- Global Performance: This combines all the per-query performance values in a single metric.
- Memory Consumption: This shows the usage of the main memory during query execution.

4.1.2 Distributed RDF Repositories

In the earliest works on query federation over SPARQL endpoints, the evaluation was carried out using the single RDF repository benchmark. Researchers normally partition those single repository benchmarks datasets, and then the federated engine executes the original queries over partitioned dataset. These single RDF repository benchmarks are generally proposed for single RDF stores, hence they are not suitable for distributed infrastructure.

FedBench

To the best of our knowledge, FedBench is the only benchmark that is particularly designed for federation over SPARQL endpoints. In order to represent real world datasets, it uses 12 Linked Open Datasets both cross-domain and in the life science domain. The cross domain dataset includes DBpedia[2], NyTimes News[3], Geonames[4], Jamendo[5], and LinkedMDB[6], while

[1]DBLP is a computer science bibliography dataset `http://dblp.uni.tier.de`
[2]DBpedia: `http://dbpedia.org`
[3]NyTimes: `http://www.nytimes.com/`
[4]Geonames: `http://www.geonames.org`
[5]Jamendo: `http://jamendo.org`
[6]LinkedMDB: `http://linkedmdb.org`

the life science dataset consists of DBPedia Drug, KEGG Compounds[7], KEGG Enzymes[8], KEGG Drugs[9], KEGG Reactions[10], Drugbank and ChEBI[11]. The selection of the datasets is based on three criterias, namely: the dataset size, coverage and schema. FedBench also provides 16 predefined queries and two performance metrics: loading time and response time.

The state of federated SPARQL query evaluations

Federation over SPARQL endpoints benchmarking is still in its infancy. To date, no single federated benchmarking process has been globally adopted. Table 4.1 shows various methods for evaluating federated engines. The majority of datasets used in evaluation are generated from partitioning a dataset or partitioning multiple merged sources. In general, federated engines execute more than 10 queries in an evaluation. Also, we noted that response time is the primary metric in all evaluations. DARQ also uses transformation time to calculate the time needed for query planning and optimisation. FedBench is frequently used as the benchmark suite in several author's evaluations. In some evaluations, the federated engines were evaluated using BSBM. In those evaluations, the BSBM dataset was divided into smaller datasets.

4.2 Proposed Performance Metrics

Based on our survey in Chapter 3 and our findings about the existing evaluations, we now propose the suitable performance metrics for assessing federated engines and SPARQL endpoints.

4.2.1 Basic Metrics

To begin with, in Figure 4.1, we describe the federated SPARQL query infrastructure and metrics that are associated to each of its components. We define five sections in which we can measure the basic performance metrics:

Section A At the client side, metrics include memory usage (primary and secondary memory) and CPU usage while sending a query, the time when the query was sent and the results

[7]KEGG Compounds: `http://www.genome.jp/kegg/compound/`
[8]KEGG Enzymes: `http://ec.bio2rdf.org/sparql`
[9]KEGG Drugs: `http://dr.bio2rdf.org/sparql`
[10]KEGG Enzymes: `http://rn.bio2rdf.org/sparql`
[11]ChEBI: `http://chebi.bio2rdf.org/sparql`

Engine	Dataset	Queries	Metrics	Benchmark System	Reference
ADERIS	DBPedia (4 sources, splitting by download name)	20	Response Time	-	[Kikuchi et al., 2010]
Anapsid	LinkedSensorData-blizzards[a], LinkedCT[b], DBPedia	30	Number of retrieved results, Response Time	-	[Acosta and Vidal, 2011]
Avalanche	IEEE, DBLP and ACM (5 sources, Merging and splitting their origin and chronological order)	3	Number of retrieved results, Response Time	-	[Basca and Bernstein, 2010]
DARQ	DBPedia (5 sources, splitting by its type)	4	Response Time, Transformation Time	-	[Quilitz and Leser, 2008]
Distributed SPARQL	BSBM dataset (6 sources)	10	BSBM Metrics, Memory Usage, CPU Usage, Bandwidth Usage	BSBM	[Zemánek and Schenk, 2008]
FedX	FedBench dataset	16	Number of selected sources, Response Time	Fedbench	[Schwarte et al., 2011]
GDS	FedBench and BSBM datasets	28	Response Time, CPU usage, Network usage, Memory usage	FedBench, BSBM	[Wang et al., 2011a]
SemWiq	KSO Sunspot[c](2 sources),Univ Graz Publication,H Alpha Exposure[d] ,Scientist[e]	8	Response Time	-	[Langegger et al., 2008]
Splendid	FedBench dataset	16	Number of selected sources, Response Time	FedBench	[Görlitz and Staab, 2011]

Table 4.1: The list of existing evaluations of a federated SPARQL query

[a] `http://wiki.knoesis.org/index.php/LinkedSensorData`

[b] `http://linkedCT.org`

[c] Kanzelhhe solar observatory as part of the Austrian Grid project, a national research project funded by the Austrian Federal Ministry for Education,Science and Culture.

[d] contains pictures taken with a special H-alpha telescope

[e] contains personal information about scientists working at the organization

received. It is hard to measure performance in the client side because this section is not under our control. However, we can calculate the amount of data delivered to the federated engine once this data is received at the federated engine side.

Section B The networking infrastructure between the client and the federated engine is the main component. One of the metrics that can be quantified in this area is the network speed and capacity. Like section A, section B is not under our control, therefore we do not consider measuring metrics in this section.

Section C Federated engine performance is normally measured in most evaluations. The metrics include memory usage, CPU usage and network usage during query execution. Although a federated engine does not store any data, it does sometimes consume secondary memory for storing the caching data and intermediate results. In addition, the main memory usage and CPU usage may increase while executing a query. The network usage can be calculated as the total volume of data transmission between the clients, the federated engine and the SPARQL endpoints. Furthermore, like the prior evaluation explained in Table 4.1, the response time and the query throughput are also two main metrics in this area.

Section D The essential difference between a single RDF repository and distributed repository is the presence of sections D and E. Similar to section B, this section consists of networking infrastructure, but it connects the federated engine and several SPARQL endpoints. We also do not take into account metrics in this section because this section is not under our coverage.

Section E This section is normally out of our control, unless the SPARQL endpoints are at the same host with the federated engine. If the SPARQL endpoints are located in the same machine, we can measure the CPU and memory usage that are required for executing sub queries at the SPARQL endpoints.

4.2.2 Extended Metrics

Apart from the basic metrics above, we next develop other metrics that are suitable for measuring the performance of framework for federation over SPARQL endpoints. Based on observing the federated engine components in Figure 3.4, there are two types of federation performance metrics: independent metrics and semi-independent metrics.

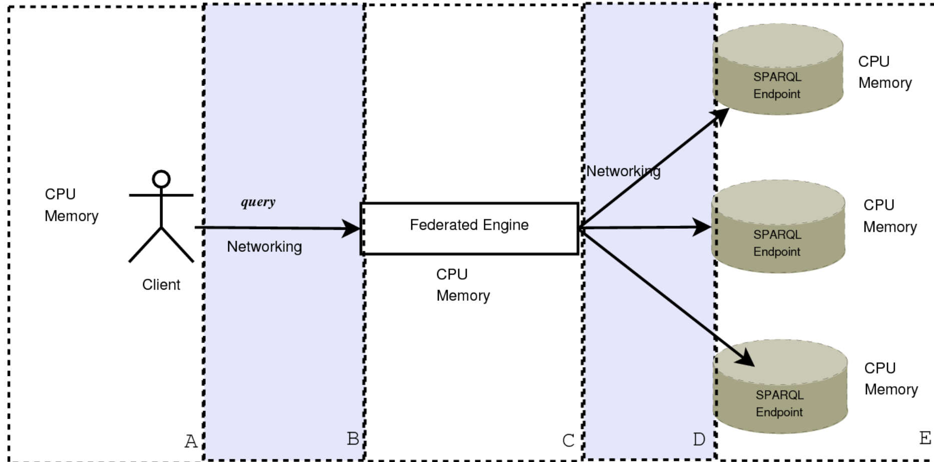


Figure 4.1: The section of infrastructure for assessing query federation framework

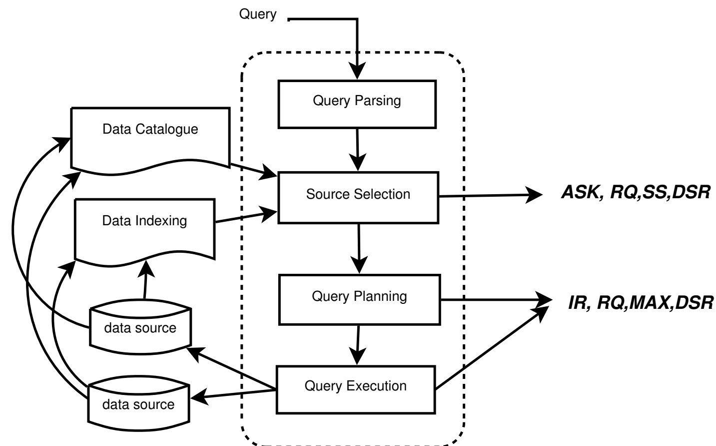


Figure 4.2: The relationship between federated engine components and independent metrics.

1. **Independent Metrics**

   Except for the network usage, the above metrics are dependent on the federation environment such as the CPU speed, memory capacity, applications etc. For instance, response time is one of the metrics that is highly related to CPU speed and memory capacity. Consequently, it is difficult to generate a gold standard for federation over SPARQL endpoints benchmarks by using such metrics. But as described in Chapter 1, we need a gold standard for a federated SPARQL query evaluation. Herein, we identify a set of independent metrics that evaluate the federated engine performance by observing the federated engine components and data transaction between the federated engine and SPARQL endpoints. We focus on the data transaction because query execution is influenced by network in the federated SPARQL query.

   Three types of data metric units arise during query execution: rows, requests and bytes. Those data metric units can represent the cost of communication between the federated engine and SPARQL endpoints. Based on the data transactions between the federated engine and the SPARQL endpoints, we identify the following independent metrics:

   - **Number of ASKs (ASK)** Several approaches deliver an ASK SPARQL query to find the relevant sources for certain subqueries (Section 3.4). The basic idea of this strategy is to discover the relevant sources with the minimum communication cost since the SPARQL `ASK` query only returns a Boolean value. Consider the example query in Listing 3.3, executed on a federation over SPARQL endpoint containing two SPARQL endpoints: Disease and Dailymed datasets. Suppose the federated engine uses SPARQL ASK query in source selection so that it sends a list of ASK queries for each triple pattern as shown in Listing 4.1. In this example, the number of ASKs will equal to the number of triple patterns times the number of SPARQL endpoints $= 4 \times 2 = 8$ requests. This metric was used by [Hose and Schenkel, 2012].
   - **Number of requests (RQ)** refers to how many SPARQL queries (`ASK`, `SELECT`, `CONSTRUCT`, `DESCRIBE`) are delivered by a federated engine to the SPARQL endpoints. Coming back to the previous example of the number of ASKs, after sending a list of `ASK` queries, the federated engine delivers the first three triple patterns to the Diseasome SPARQL endpoint and the fourth triple pattern to the Dailymed SPARQL endpoint (Listing 4.2). As a result, the number of requests

for executing that query is the sum of the number of `ASK`s and the number of `SELECT`s, which in this case is: $8 + 2 = 10$ requests. RQ was used for assessing performance of [Hose and Schenkel, 2012; Schwarte et al., 2011; Montoya et al., 2012b; Görlitz and Staab, 2011].

- **Size of the intermediate results (IR)** The intermediate results are all answers retrieved by the federated engine from all SPARQL endpoints during query execution. The size of the intermediate results can be predicted by cardinality estimation at the query planning stage. To compute the size of the intermediate results, we count the total number of rows received by the federated engine at runtime. Given two SPARQL endpoints, Diseasome and Dailymed that store datasets as in Figure 2.5 and the query in Listing 3.3 received by the federated engine, then the federated engine receives a row from the Diseasome SPARQL endpoint after executing the first three triple patterns. In addition, it also retrieves a row from the Dailymed SPARQL endpoint. In total, the size of intermediate results to execute the query in Listing 3.3 is two rows. This metric was used by [Montoya et al., 2012b].
- **Maximum Results (MAX)** is defined as the maximum size of intermediate results obtained at runtime per query request. Since both Diseasome and Dailymed SPARQL endpoints receive a row, the maximum results is one row.
- **Amount of data sent and received (DSR)** Apart from calculating the number of rows, we also consider data transmission in byte unit. Current of federated engines only estimate the join cardinality based on the number of triples. In fact, the number of rows cannot reflect the real quantity of data transmission. Different rows have different size in bytes. A literal object could contain more characters than a URI object. Various query forms are transmitted from a federated engine to SPARQL endpoints. Consequently, the federated engine sends different amount of data based on the number of characters used to formulate a SPARQL Query. For a simplified instance, regardless of the HTTP header size, we can calculate the total amount of data sent and received as the number of characters in all `ASK` queries (as in Listing 4.1) and subqueries and the answers of those subqueries (as in Listing 4.2).

Besides the data transmission calculations, we also consider the **Number of Selected Sources (SS)** proposed by FedX and Splendid as one of the independent metrics. SS

Listing 4.1: List of ASK queries sent to determine relevant sources for query in Listing 3.3

```
ASK {
        ?disease a diseasome:diseases .
}

ASK {
        ?disease rdfs:label ?diseasename .
}

ASK {
        ?disease diseasome:possibleDrug ?drug .
}

ASK {
        ?drug dailymed:name ?drugname .
}
```

refers to the number of SPARQL endpoints involved to accomplish a single query execution. A query can be answered by either partial or all sources. The effectiveness of the source selection strategy can be shown from the number of SPARQL endpoints accessed. Since it is hard to distinguish which queries are part of the source selection process and which ones are not, we ignore the `ASK` queries delivered to SPARQL endpoints.

All independent metrics are influenced by the strategy applied at the federated engine components. We find a relationship among the independent metrics and federated engine components as depicted in Figure 4.2. Obviously, the number of ASK's and SS's depend on the source selection approach. The IR and MAX values are affected by the query optimisation approach for query planning and execution. DSR and RQ are influenced by the source selection, query planning and execution strategies since data transmission between the federated engine and SPARQL endpoint happens during source selection and query execution, .

2. **Semi-Independent Metrics**

   As part of the federated engine, a SPARQL endpoint can influence the performance of a framework for federation over SPARQL endpoints. However, most of the current evaluation approaches disregard the existence of the SPARQL endpoint. Hence, the query optimisation in the federation engine only focuses on getting results quickly

Listing 4.2: List of sub queries sent to SPARQL endpoints to accomplish query in Listing 3.3

```
SELECT * {
        ?disease a diseasome:diseases .
        ?disease rdfs:label ?diseasename .
        ?disease diseasome:possibleDrug ?drug .
}

SELECT *
{
        ?drug dailymed:name ?drugname .
}
```

without considering the SPARQL endpoint capability. For instance, FedX (see at Section 3.5.2) proposed the bound join strategy to merge a number of intermediate results variables in a single query. In theoretical, ideal conditions, this can reduce the number of requests and size of intermediate results. However, each request may consume significant bandwidth to retrieve the results from SPARQL endpoint. Moreover, the federated engine might be able to communicate more than one request to a SPARQL endpoint in a period of time. Consequently, the SPARQL endpoint workload tends to be high. In order to ensure the sustainability of a SPARQL endpoint server, the SPARQL endpoint server sometimes rejects an expensive query and only returns a limited number of results. Ultimately, the query answer could be incomplete as described in Figure 1.4 and in the worst case, the federated engine can fail to finish the query execution. Based on observing the capability of a SPARQL endpoint, we generate semi-independent metrics using independent metrics as the variables:

- **Request Workload (*RW*)** Typically, a public SPARQL endpoint does not allow us to send many requests in an interval of time. Hence, we define a request

workload (RQ) as — the approximation of the number requests delivered to a number of SPARQL endpoints in a given interval time – which can be formulated as follows:

$$RW = \frac{RQ}{T * SS} \tag{4.1}$$

where T is the response time, RQ is the number of requests and SS is the number of SPARQL endpoints selected[12]. Coming back to the previous example of the number of requests(RQ) and the number of sources selected(SS) in the independent metrics section, RQ is 10 requests and SS is 2 sources. Given T is 60 seconds, then $RW = \frac{10}{60*2} = 0.833$

- Average Answer Size (ANS) A SPARQL query can return zero or many rows. In order to decrease bandwidth usage, it normally truncates answer size to a certain number of rows. ANS indicates the capacity of a SPARQL endpoint to answer a query, which can be calculated as follows:

$$ANS = \frac{IR}{RQ} \tag{4.2}$$

where IR is the size of intermediate results and RQ is the number of requests. As described in the previous example of the size of intermediate results, Listing 3.3 generates two rows. Based on that example, we can calculate $ANS = \frac{2}{10} = 0.2$

- **Average Data Received (ADR)** is defined as the amount of data received by a federated engine in bytes per query request. The high value of ADR implies a costly communication between the federated engine and the SPARQL endpoints.

$$ADR = \frac{DSR}{RQ} \tag{4.3}$$

3. **Composite Metrics**

As shown in Table 4.1, most evaluations execute more than 10 queries. It is difficult to judge which federated engine performs better than others by looking at multiple results. For the sake of readability, we aggregate the results of each performance metric into a single value. In order to avoid trade-offs among queries, we assign a weight to each query using the variable counting strategy from ARQ Jena [Stocker and Seaborne, 2007]. This weight indicates the complexity of the query based on the

[12]This modifies our previous request workload formula

selectivity of the variable position and the impact of variables on the source selection process. The complexity of a query can influence the federation performance. Hence, we normalise each performance metric result by dividing the metric value by the weight of the associated query. In the context of federated SPARQL queries, we set the weight of the predicate variable equal to the weight of the subject variable since most of the federated engines rely on a list of predicates in order to select the data location. Note that, a triple pattern can contain more than one variable. The choice of the weight of the subject variable w_s, predicate variable w_p and object variable w_o for the triple pattern τ can be explained as follow:

$$w_s(\tau) = \begin{cases} 3 & \text{if the subject of triple pattern } \tau \in V \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

$$w_p(\tau) = \begin{cases} 3 & \text{if the predicate of triple pattern } \tau \in V \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

$$w_o(\tau) = \begin{cases} 1 & \text{if the object of triple pattern } \tau \in V \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

Finally, we can compute the weight of query q:

$$weight(q) = \sum_{\forall \tau \in q} \frac{w_s(\tau) + w_p(\tau) + w_o(\tau) + 1}{MAX_COST} \tag{4.7}$$

where $MAX_COST = 8$ because if a triple pattern consists of a set of variables that are located in all three positions, the weight of the triple pattern is 8 (3+3+1+1).

Given that Q is a set of queries q in the evaluation, we then define the weight normalisation of the query q ($\omega(q, Q)$) in query set Q as follows:

$$\omega(q, Q) = \frac{weight(q)}{\sum_{qt \in Q} weight(qt)} \tag{4.8}$$

By using the weight normalisation of a query, we can align the query performance results afterwards. Each result is calculated individually. Given that Q is a set of queries q in the evaluation and that m is a set of performance metric results associated with the query set Q, then the final metric μ for the evaluation is defined in Equation

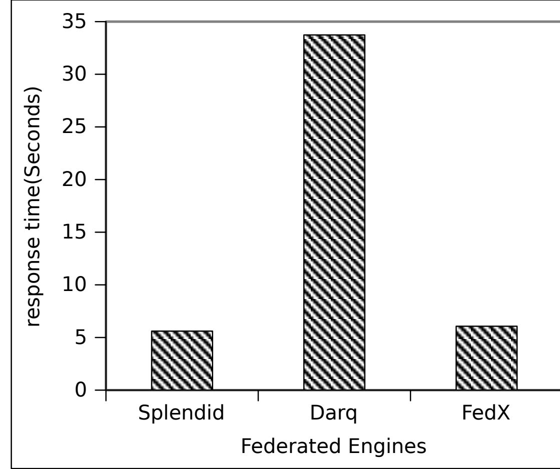


Figure 4.3: Response Time (seconds).

4.9.

$$\mu(Q, m) = \sum_{q \in Q} m_q \omega(q, Q) \tag{4.9}$$

Given an evaluation with two queries and two response time results as follows:

Query 1 (15 seconds): `select * { ?s ?p ?o .}`
Query 2 (20 seconds): `select * { ?s rdfs:label ?o . ?s diasesome:size 1 . }`

The weight of Query 1 (q_1) is $\frac{3+3+1+1}{8} = 1$, whereas weight of query 2 (q_2) is $\frac{3+1+1}{8} + \frac{3+1}{8} = 1.125$. Now, the weight normalisation of two queries can be computed: $\omega(q_1, Q) = \frac{1}{1+1.125} = 0.47$, $\omega(q_2, Q) = \frac{1.125}{1+1.125} = 0.529$. The final metric for response time is $0.47 * 15 + 0.529 * 20 = 17.63$ seconds.

4.3 Experiment and Result

In order to investigate the performance of existing query federation frameworks from different metrics, we perform a comprehensive experiment on Linux Ubuntu 64 bits. Three SPARQL endpoint servers and federated engine are set up on a single machine. Dailymed dataset is divided (Table 4.2) based on its classes: Drugs, Ingredients and Organizations and store them in different SPARQL Endpoints. We ran each of 16 queries detailed in Appendix B.1 three times, once on FedX [Schwarte et al., 2011], Splendid [Görlitz and Staab, 2011] and

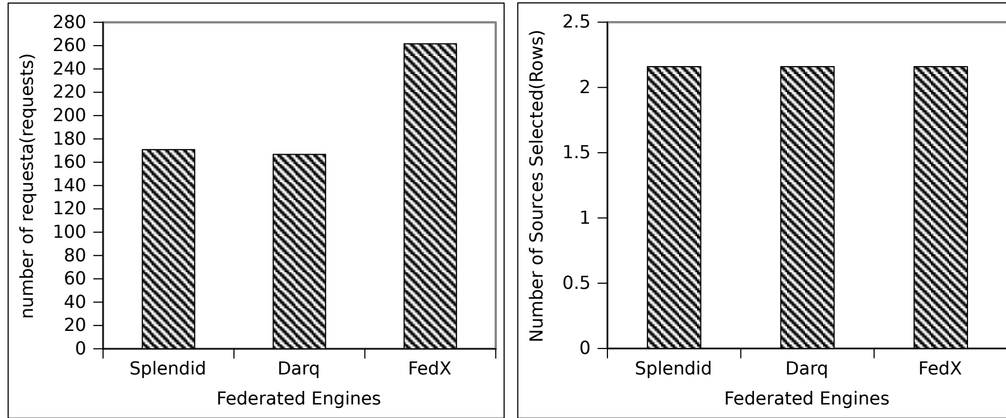
Source-set	Triples	Classes	Properties	Subjects	Objects
Drugs	147245	2	26	4308	56973
Ingridients	11212	1	3	4066	4978
Organizations	5773	1	4	711	5055

Table 4.2: Source-set Used in Evaluation

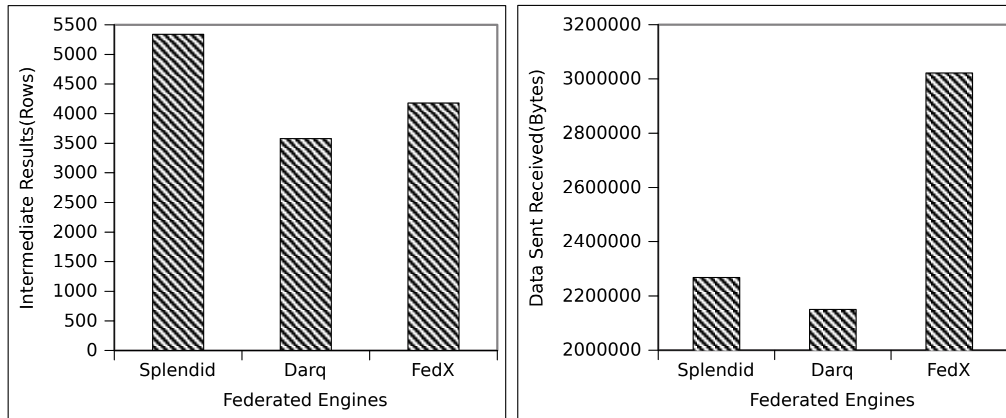
DARQ [Quilitz and Leser, 2008] federation engines. The final results shown are the composite metrics of independent and semi-independent metrics.

DARQ failed at Queries 4 and 5 since these queries contains unbound predicate, while FedX could not perform Query 6 because of the evaluation time out issue. In general, the fastest response time was achieved by Splendid and FedX (Figure 4.3), but FedX communication was expensive (Figure 4.4(d)). Splendid and DARQ produced high intermediate results (Figure 4.4(c)), but the volume of data transmission was low (Figure 4.4(d)). All federated engines have a source selection strategy before sending the sub queries, therefore the number of selected sources in all queries are the same (Figure 4.4(b)). FedX does not have a data catalogue to predict the data location, consequently, the number of requests was higher than the other federated engines (Figure 4.4(a)). Only Splendid and FedX generated ASK queries during execution (Figure 4.4(e)). As seen at Figure 4.5(b) and 4.5(a), the lowest of ADR and ANS were obtained by FedX, since it uses a bound join strategy. As a result, the FedX request workload is lower than the DARQ and SPLENDID request workload (Figure 4.5(c)). The speed of a federated engine can be a main indicator of the framework for federation over SPARQL endpoint performance, but other metrics should be considered to measure the framework performance. For instance, even though FedX can answer a query quickly, it produced too many requests which can cause poor performance of SPARQL endpoints. The communication cost can not only be described by the average size of intermediate results, but the average data received should also be taken into account. As shown in Figure 4.5(b) and 4.5(a), SPLENDID had more intermediate results than DARQ, but it received less data than DARQ. Thus, it is necessary to calculate the cost of query execution based on the size of the data, not only based on the cardinality estimation.

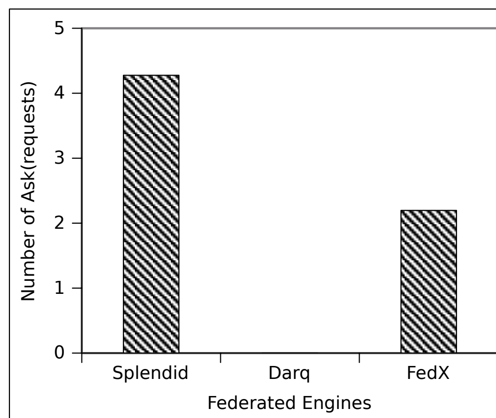
To provide a better comparison amongst federated engines, we combine the response time, the data sent received and the number of requests by using the geometric mean since the small value produced by these metrics indicates better performance of a federated engine. As seen in Figure 4.6, the best performance was achieved by SPLENDID, followed by FedX and DARQ. Using a data catalogue and the ASK SPARQL query does minimise the bandwidth usage and number of requests between SPLENDID and the SPARQL endpoints.



(a) Number of Requests(RQ) (b) Number of Sources Selected(SS)

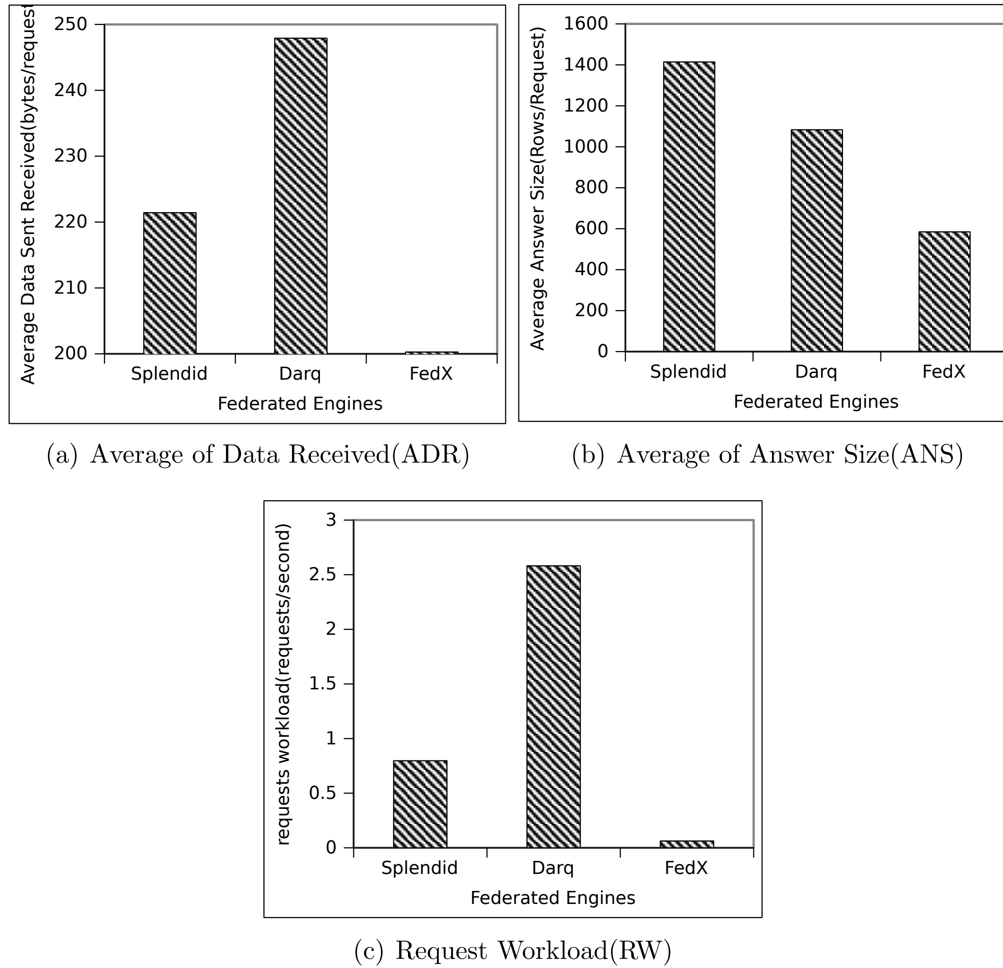


(c) Size of Intermediate Results(IR) (d) Amount of Data Sent and Received(DSR)



(e) Number of ASKs(ASK)

Figure 4.4: Performance of the three federated engines measured by independent metrics



(a) Average of Data Received(ADR)

(b) Average of Answer Size(ANS)

(c) Request Workload(RW)

Figure 4.5: Performance of the three Federated Engines Measured By Semi-Independent Metrics

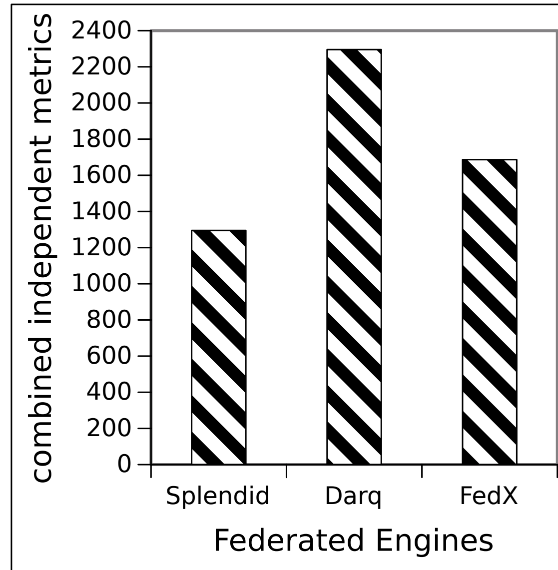


Figure 4.6: Combined Indepedent Metrics Results

4.4 Conclusion

We have presented a holistic evaluation of the existing federation SPARQL Engines by introducing three types of metrics : independent metrics, semi-independent metrics and composite metrics. The independent metrics are not influenced by the evaluation environment such as the hardware and software. These metrics include the number of requests, number of ASKs, size of intermediate results, amount of data sent and received and number of selected sources. Most of those metrics are obtained from the data transmission between the federated engine and the SPARQL endpoints. We also proposed three metrics that are associated with SPARQL endpoint capacity, namely the Request Workload, the Average Answer Size and the Average Data Received.

Based on these various metrics, a user can select the metrics that are suitable for his environment and use case. On one hand, the speed of a federated engine to answer a query can be a main indicator of the federation performance, but on the other hand, sometimes this leads to more expensive communication such as a high number of requests delivered in given interval of time and high data transmission. In real cases, such conditions can significantly impact the SPARQL endpoint performance. Eventually, it will also affect the whole federation system. Hence, a federated engine developer should consider approaches for minimising the number of requests such as applying a window size in a query execution strategy.

Federated engines normally apply the cardinality estimation based on how many rows selected which is more suitable for the single RDF query optimisation. Since the federation framework encounters a network communication issue, a weighting function should be assigned to objects with literal values. Literal values especially strings, might consume more bandwidth than URI values.

Chapter 5

The Cost And Benefit of Exploiting Links in Federated SPARQL Query*

In this chapter, firstly we describe our motivating example to carry out an investigation that explores the benefits and the costs of using links in a federated SPARQL query. Next, we define the link term in a federated SPARQL query. Continuing, we explain the cost and benefits of links on the performance of a federated SPARQL query engine. In the final section, we conduct three experiments to analyse the impact of links on federated SPARQL query.

5.1 Motivating Example

Different initiatives have been taken by researchers to link data both at schema and instance level to facilitate efficient query processing. At schema level, the linking mechanism mainly relied on aligning the ontologies used by different sources e.g. BLOOMS [Jain et al., 2010]. Alternatively, [Hasnain et al., 2012] proposed a methodology to facilitate *a posteriori data integration* to help SPARQL federation by cataloguing the data. Similarly, SILK [Jentzsch et al., 2010] and LIMES [Ngomo and Auer, 2011] facilitate link generation between instances from two sources based on matching the predicates. The interlinking of multiple sources in the *Web of Data* enables users to navigate amongst these sources, similar to the way the users currently navigate through different web pages in the *Web of Documents*. In this way, the LOD Cloud can be significantly benefit users to retrieve a meaningful information by enhancing existing tasks such as querying, reasoning, and knowledge discovery [Jain et al.,

*Parts of this chapter have been published as [Rakhmawati et al., 2014a]

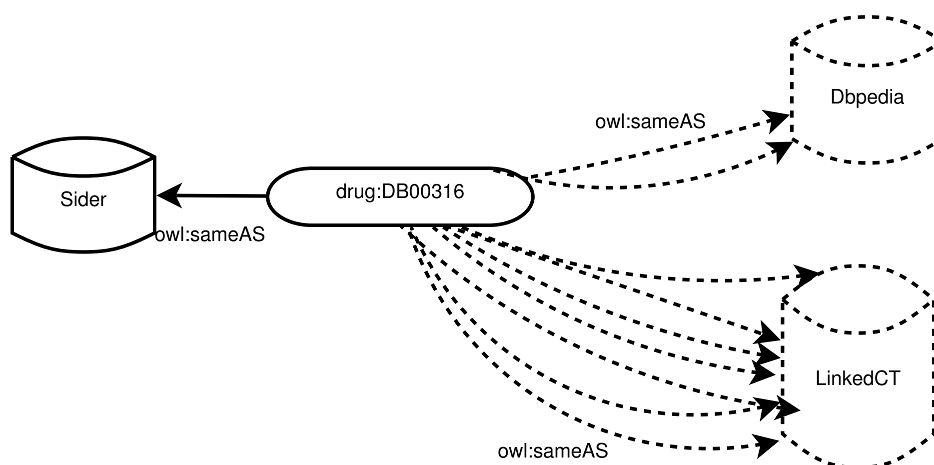


Figure 5.1: The example of the hidden cost of links

2010].

Although the interlinks amongst sources have several benefits associated with them for the data users, there may have been some hidden costs associated with federating a query over multiple SPARQL endpoints. For example, a single entity which is connected to multiple object values with the same link. Those objects values may not be available in any datasets for a particular federation scenario. Consequently, these links may increase the communication cost between the federated engine and SPARQL endpoints because many subqueries with those objects cannot be answered by the SPARQL endpoints. As a motivating example, assume that a federated engine receives Query 3.9 and drug DB00316 in the Drugbank dataset has seven `owl:sameAs` links to LinkedCT[1], one link to Sider and two links to DBPedia (Figure 5.1). If the defined federation framework does not contain DBpedia and LinkedCT datasets, these links that point to DBPedia and LinkedCT datasets can cause unexpected intermediate results during query execution. Consequently, these intermediate results can increase the data transmission between the federated engine and SPARQL endpoints. Hence, there is a need to investigate the impact of distinct links from one dataset to others for calculating the cost of any federation framework.

As mentioned earlier, researchers have been focusing on presenting new ideas and tools to facilitate data integration and interlinking, but no initiative has been taken (to the best of our knowledge) to analyse the costs and benefits of link i.e how the costs of a federated SPARQL query change while accessing the data from multiple locations having links between them and in compare to the cost associated when accessing the data without links.

[1] `http://linkedct.org/`

5.2 Related Work

There have been studies carried out to the benefit of interlinking. [Hausenblas, 2009] illustrates how the link can improve the web application development. With respect to the benefits of links for specific domains, the link is useful for discovering musics [Raimond et al., 2009], drugs [Jentzsch et al., 2009] and experts [Stankovic et al., 2010]. [Iqbal and Hausenblas, 2013] analysed the benefits of link in the Open Source Software development. LIDAQ [Umbrich et al., 2012a] investigate the benefit of `owl:sameAs` and `rdfs:seeAlso` links in terms of increasing the recall of the querying results of a link traversal infrastructure. Along these lines, our work mainly concerns in the benefits of the links in the context of federated SPARQL query.

Apart from the advantages of links, several initiatives developed the cost model for distributed SPARQL query [Obermeier and Nixon, 2011] and federated SPARQL query [Rakhmawati, 2013b]. [Obermeier and Nixon, 2011] builds a distributed query cost model using the R system, however it does not provide an evaluation for the cost model. Our initial work to build a cost model for a federated SPARQL query is presented by [Rakhmawati, 2013b]. We formulated a cost model based on the multiple regression models where the performance metrics of the federated engine is the dependent variable and the dataset metrics is the independent variables. In this thesis we do not build a cost model for distributed SPARQL query in general, but we observe the costs of the existence of the links in a sense of querying data over SPARQL endpoints.

5.3 Interlinking in Federation over SPARQL Endpoints

Federation over SPARQL endpoints consists of a federated engine as the mediator and a group of autonomous SPARQL endpoints. Although each SPARQL endpoint is independent, they could be virtually interlinked with several links which connect entities amongst SPARQL endpoints as shown in Figure 5.2. In this figure, the Diseasome, the Sider and the Drugbank is interconnected by `owl:sameAs` link.

For a better understanding, we initially define links that can merge data from two or more datasets. Two or more sources are linked if there is a path from one source to other sources. The path can be formally defined as follows:

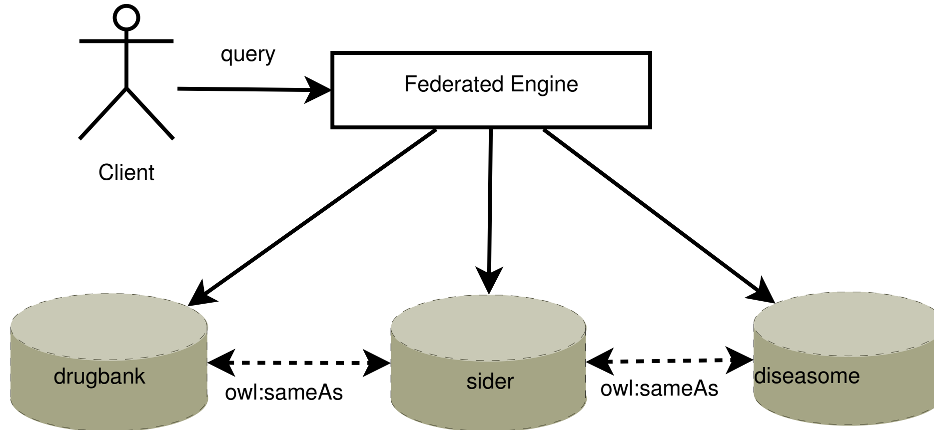


Figure 5.2: A sample of a federation over SPARQL endpoints where some entities in SPARQL endpoint are interlinked with `owl:sameAs`

Definition 5.1 *Let D be a source set. The set of all $paths(D)$ is defined inductively as follows:*

1. *$\langle a, b \rangle$ is $paths(D)$ if there exists $d \in D$ and $a, p \in U$ such that $(a, p, b) \in d \vee (b, p, a) \in d$.*
2. *if there is a path $\langle a_1, ..., a_n \rangle$ in $paths(D)$ and if there is $d \in D$ and $b, p \in U$ such that $(a_n, p, b) \in d \vee (b, p, a_n) \in d$ then $\langle a_1, ...a_n, b \rangle \in paths(D)$.*
3. *if there is a path $\langle a_1, ..., a_n \rangle \in paths(D)$ and there is a resource b such that $(a_n, owl{:}sameas, b)$ is in the transitive symmetric closure of all owl:sameAs statements of union d in D then $\langle a_1, ..., a_{n-1}, b \rangle \in paths(D)$.*
4. *these are all the paths.*

Given $paths(D)$, now we can determine whether two sources are linked.

Definition 5.2 *Two sources $R, S \in D$ are linked if there is a path $\langle a_1, ...a_n \rangle \in paths(D)$ such that there is $a_i \in R \wedge a_j \in S$ with $1 <= i, j <= n$.*

Let R be a SPARQL endpoint that contains the following triples:

```
exR:Alice rdf:type foaf:person .
exR:Alice owl:sameAs exS:student123 .
```

and let S a also be SPARQL endpoint that contains the following triples:

```
exS:student123 rdf:type exS:student .
```

Since there is a path exR:Alice-exS:student123-exS:student from R and S, we can say that R and S are linked.

The last condition of Definition 5.1 is intended for covering the characteristic of `http://www.w3.org/2002/07/owl#sameAs` (owl:sameAs). Owl:sameAs is a link that connects two entities that are located in different locations but have the same characteristics. This link is one of popular links that integrate data from multiple locations.

If we add exS:student123 owl:sameAs exR:Alice in S SPARQL endpoint as shown in the following triples:

```
exS:student123 rdf:type exS:student .
exS:student123 owl:sameAs exR:Alice .
```

then the closure between (exR:Alice owl:sameAs exS:student123) $\in R$ and (exS:student123 owl:sameAs exR:Alice) $\in S$ is symetric.

Next, if we let R be a SPARQL endpoint stores the following triples:

```
exR:Alice rdf:type foaf:person .
exR:Alice owl:sameAs exT:Alicia .
```

and S be a SPARQL endpoint has the following triples:

```
exS:student123 rdf:type exS:student .
exS:student123 owl:sameAs exT:Alicia .
```

We can conclude that R and S are also linked because the relation between (exR:Alice owl:sameAs exT:Alicia) $\in R$ and (exS:student123 owl:sameAs exT:Alicia) $\in S$ is transitive closure. The compositions of transitive closure between R and S is {(exR:Alice owl:sameAs exT:Alicia), (exR:Alice owl:sameAs exS:student123), (exS:student123 owl:sameAs exT:Alicia)}. If ext:Alicia cannot be found in any source that is part of source-set D, R and S might be also linked to another source.

In this chapter, we analyse the effects of the existence of the links since links have an important role when merging data from different data-sources. We only consider observing links that are produced by data publisher for connecting two entities that are located in two locations. Such links are predicates that connect two sources with length of path is two. For instances, owl:sameAs, rdf:seeAlso, etc. In early phase of distributed RDF data, these links are used to navigate from one source to another source. And at present, there are more links generated to create Linked Open Data (LOD) cloud.

Listing 5.1: Drugbank Sample Dataset

```
1 drugbankdrugs:db00316 rdf:label "Acetaminophen" .
2 drugbankdrugs:db00316 owl:sameAs sider:1983 .
3 drugbankdrugs:db00316 drugbank:target drugbanktargets:290 .
```

For example, consider Listing 5.1 which contains triples related to *Acetaminophen* and given a dataset D as shown in Figure 3.2 that comprises: Drugbank, Dailymed, Sider and Diseasome. According to Definition 5.1, `rdfs:label` in the first triple is not a link because its object type is a literal. `owl:sameAs` is one of links because `sider:1983` is a subject in Sider dataset. In contrast, `drugbank:target` is not a link since the subject and object is located at the same location.

5.4 The Cost and Benefit of the Links

The existence of links can replace the objects similarity (Section 3.3.3) approach in some cases. In order to compare the benefits of exploiting links and other approaches (without any links) for federated SPARQL execution strategies, we use the RDF selectivity estimation. In our scenario, we assume that each link and each predicate has exactly one object value in order to produce the same number of intermediate results. For the sake of simplicity, we only consider the two triple patterns that combines data from more than one dataset regardless of the rest of query. Basically, the object similarity and the reusing of an identifier approaches are object-object join pattern. Suppose that $?x_1p_1?y_1$ is a triple query pattern that matches data-source d_1 while $?x_2p_2?y_2$ is a triple query pattern that matches data-source d_2. Then an object-object join pattern can be defined as: $[?x_1p_1?y_1]$ and $[?x_2p_2?y_1]$, whereas the using link approach is a subject-object join pattern which can be written as $[?x_1p_1?y_1]$ and $[?y_1p_2?x_2]$. These joining strategies cost are influenced by the execution strategies. In general, there are two main types of execution strategies in the federated SPARQL query namely *Nested-loop join* and *Bind join* [Quilitz and Leser, 2008] which have been explained in Section 3.4.5. In the Nested-loop join scheme, the federated engine sends each subquery to all SPARQL endpoints simultaneously and joins the results locally. By contrast, in order to minimise the number of intermediate results, federated engines apply the Bind join strategy by executing each subquery one by one and assigning the results from the previous subquery as a constraint on the next subquery.

In the case of object-object join query pattern, the Nested-loop join generates different

intermediate results from the Bind join. Suppose that $T(p_1)$ is a function to count the number of triples that match to pattern $[?x_1 p_1 ?y_1]$ while $T(p_2)$ is a function to count the number of triples that match to pattern $[?x_2 p_2 ?y_1]$. Then, the intermediate results of object-object join pattern in the Nested-loop join strategy is $T(p_1) + T(p_2)$. Bind join produces fewer number of intermediate results because $?y_1$ from the previous subquery becomes a constrains for $[?y_1 p_2 ?x_2]$. Let $T(p2, y1)$ be a function to calculate the number of triples that match pattern $[?x_2 p_2 ?y_1]$ where $?y_1$ is replaced by the previous subquery result, then the intermediate results of object-object join pattern in the Bind join strategy is $T(p_1) + T(p_2, y_1)$. However, by contrast, using links that implements the subject-object join pattern produces the same result in both the Nested-loop join and the Bind join strategies.

According to [Stocker et al., 2008], the subject-object join pattern is more selective than the object-object join pattern. Hence, the existence of links could provide more benefits to a federated engine that applies the Nested-loop join strategy. In order to support the argument made above, we conducted an experiment which is detailed in Section 5.5. Apart from that, there are also several potential benefits of links in the federated SPARQL query:

- As described in Section 3.3, without using links, we can use objects similarity to query over multiple sources. However, the objects similarity often encounters case sensitivity problem due difference in the capitalisation of the entity label. To tackle the problem of case sensitive, we can use the `FILTER` keyword to compare object values. In addition, a `REGEX` keyword can also be added. However, these two keywords may increase the cost of a query execution. Apart from case sensitivity, typographical errors in writing a label can potentially lead to decreases the number of query results. The usage of links at a federated SPARQL query can avoid these issues.

- The links generation requires the domain experts to curate the validity of links. Hence, the existence of these links can increase the accuracy of federated SPARQL query results. However, other strategies such as object similarity may return invalid results or fewer results. This statement is supported by experiment 3 in Section 5.5 which compares the accuracy of results between queries containing links and those without any links.

In the following section, we will describe our experimental set-up to investigate the costs and benefits of interlinking in the context of a federated SPARQL query.

Dataset	Properties	Triples	Links
Dailymed	28	164276	39635
Drugbank	118	766920	56958
Disease	19	91182	31750
Sider	11	193249	20294

Table 5.1: Statistic for the dataset used in the evaluation

5.5 Experimental Setup

We carried out three experiments with three different objectives: 1) investigate the effects of the existence of links at federated SPARQL query 2) observe the impact of number of links on the performance of a federated engine and 3) study how the presence of links contributes to the accuracy of the results. We run these experiments on an Intel Xeon CPU X5650, 2.67GHz server in which Ubuntu Linux 64 bit is installed as the Operating System and Fuseki 1.0 as the SPARQL endpoint server. For each dataset, we set-up a Fuseki server which is distinguished by unique port. The execution time-out for each query is set to one hour. Our experiment uses four Life Science domain datasets which are related to the pharmaceutical domain (Figure 3.2), namely Dailymed, Drugbank, Disease and Sider. Further statistical details of each dataset can be found in Table 5.1.

The details of our experiments are explained as follows:

1. The first query set consists of six objectives. Each objective contains two queries: one uses links while the other uses object similarity (Section 3.3.3). The aim of having two queries for each objective is to compare between a query that exploits links and a query that does not use any links. Since we can not find the object similarity case for the Diseasome dataset, first query set only uses Drugbank, Dailymed and Sider datasets. Our queries for this experiment can be found in Appendix B.2. The first query triple pattern in each query always contains a constant subject in order to avoid different number of results in the same query objective. Note that, we only run the first experiment on DARQ, Splendid and FedX because we want to investigate the impact of the links with three different execution strategies. Although we earlier described five federated SPARQL query approaches in Section 3.3, we only designed query set based on two approaches: the object similarity and using link approaches because these two approaches address the same objective in two different ways. Other approaches with the same objective, such as using the `SERVICE keyword` use either of the two approaches we study: object similarity and using links approaches. However, we only take into

account a query which does not explicitly contain the SPARQL endpoint address. We choose federated engines that have a source selection strategy for executing a query, namely Darq, SPLENDID and FedX. In order to assess the performance of federated engines, we use five metrics 1) Number of rows 2) Response time of the federated engine to execute a query 3) Data transmission between the federated engine and SPARQL endpoints 4) Number of intermediate results to execute a query 5) Number of requests delivered to all SPARQL endpoints.

2. The second query set comprises 210 queries including: 151 two-chains, 38 three-chains and 21 star queries. Besides the path pattern, the query set is differentiated by its classes, links and predicates. Our query set covers all classes and properties in the dataset. However, not all properties are included in our query set since each predicate is not always found in all datasets. We also design some queries to retrieve data from all data-sources. The query pattern is differentiated by its shape of path, namely the star shape, chain shape and hybrid shape. In any pattern, the property is a constant whereas the subject or object of the triple pattern could be a variable. In the query federation framework, the path query slightly differs from the centralised repository infrastructure. The path in our evaluation refers to the links between datasets, not the path between subjects and objects in a single dataset.

    - A **Chain Shape** comprises more than one triples which are connected by a link. As shown in the Figure 3.2, a chain shape example is a relation `owl:sameAs` between `sider:side_effects` with `diseasesome:diseases`. The chain shape may involve more than two datasets. For instance, the path `drugbank:brandedDrug` between `drugbank:drugs` and `dailymed:drugs` then `dailymed:drugs` also has `dailymed:possibleDiseasetarget` linking to `diseasesome:diseases`.
    - A **Star Shape** is useful for retrieving triples having the same link from multiple datasets. The star path is formed by a link to different datasets which belongs to a single subject or entity. For example, `sider:drugs` has `owl:sameAs` that links to `dailymaed:drugs` and `drugbank:drugs`. Another star shape example is `diseasome:possibleDrug` making the connections between `diseasome:diseases` and `dailymed:drugs` or `diseasome:diseases` and `drugbank:drugs`. This shape allows a federated engine to deliver the same subquery to multiple SPARQL endpoints in parallel.
    - A **Hybrid shape** is a combination of star and chain shape joining more than two

datasets.

The query sets were executed 3 times over 4 federation engines, namely DARQ, Splendid, Sesame SAIL Alibaba and FedX. With respect to the execution strategy, DARQ implements either Nested-loop join and Bind strategy, Splendid uses Bind join strategy and FedX applies the Bound join strategy. In order to see the effect of the links in various execution strategies, we set DARQ to implement the Nested-loop join. Note that, we do not use the current Sesame that already supported `SERVICE` keyword, but we run our evaluation over Sesame SAIL Alibaba which allows us to query without a `SERVICE` keyword specified.

3. The last experiment used four queries which have the same objective: *obtain a list of branded drugs from the Dailymed dataset and its generic drugs from the Drugbank dataset*. These queries can be explained as follows:

   (a) Compare the label of *inactive ingredient* of the drugs in the Dailymed dataset to the label of drugs in the Drugbank dataset.

   (b) Compare the label of *active ingredient* of the drugs in the Dailymed dataset to the label of drugs in the Drugbank dataset.

   (c) Compare the label of *active moiety* of the drugs in the Dailymed dataset to the label of drugs in the Drugbank dataset.

   (d) Exploit the `dailymed:genericDrug` link to discover the relation of the branded drug in the Dailymed and its generic drug in the Drugbank.

   We picked active ingredient, inactive ingredient and active moiety as part of the comparisons since these properties usually refer to the generic drug name. Each drug brand name consists of at least one active ingredient which refers to the generic name and many inactive ingredients (such as substances including colors, flavours, preservatives, and materials that bind to the drugs). After getting the query results, we asked two experts (a pharmacist and a cardiologist) to evaluate the accuracy of our results. We limit the validation to the top ten results. The detail of this experiment is reported in Appendix D.

A complete query sets for all experiments can be found at `http://github.com/nurainir/costbenefitlinks`.

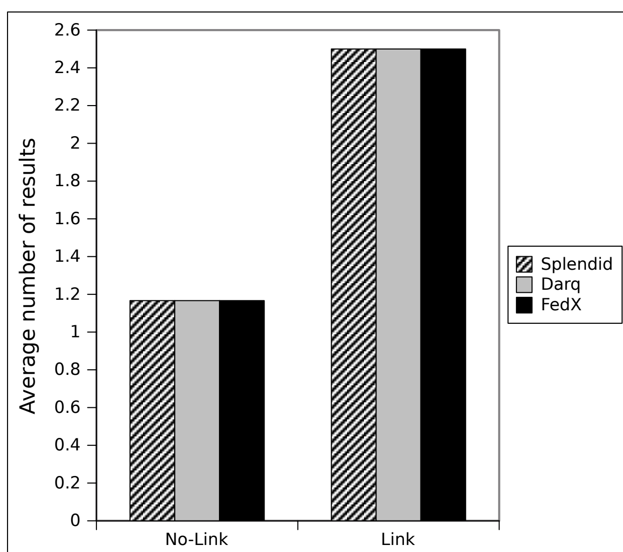


Figure 5.3: Average Number of Rows

5.6 Results and Discussion

The first experimental results are depicted in Figure 5.3-5.7 and the second experimental results are presented in Figure 5.8. As illustrated in Figure 5.3, all federated engines return the same result, but we obtained more results while using *links*. Without a link, the query execution often encounters the case sensitivity and invalid label issues. Hence, fewer results are retrieved without links than with links. In general, there are not many differences in the results (Figure 5.4-5.7) between the no-link query and the link query when these queries are executed by Splendid and FedX since these engines implement the Bind join strategy. DARQ, the only federated engine implementing Nested-loop join shows that the no-link query reduce the performance of DARQ.

In total, we have 2520 results (3 running times x 4 frameworks x 210 queries) in the second experiment but 108 results are discarded. The reason for removing results are: 1) the query execution time exceeds one hour 2) query results are below 50% of the completeness value. The success of query processing is not only defined by the completion of execution, but it also depends on the validity of the result. Thus, we consider calculating the completeness of the results after query execution is done. The completeness is defined as the ratio of the number of true answers retrieved to the number of true answers in the dataset. In order to describe the results, we calculate the arithmetic mean of multiple different queries results having 1) the same number of distinct linked datasets and 2) the same number of links.

A high number of links can lead to a high communication cost between the federated

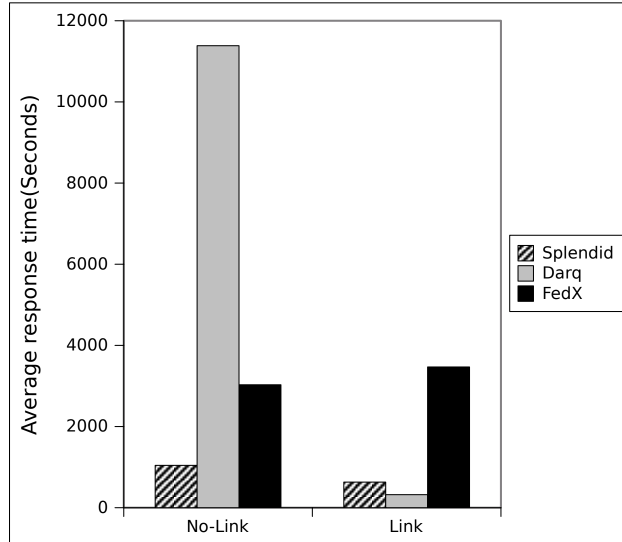


Figure 5.4: Average Response Time (in log scale)

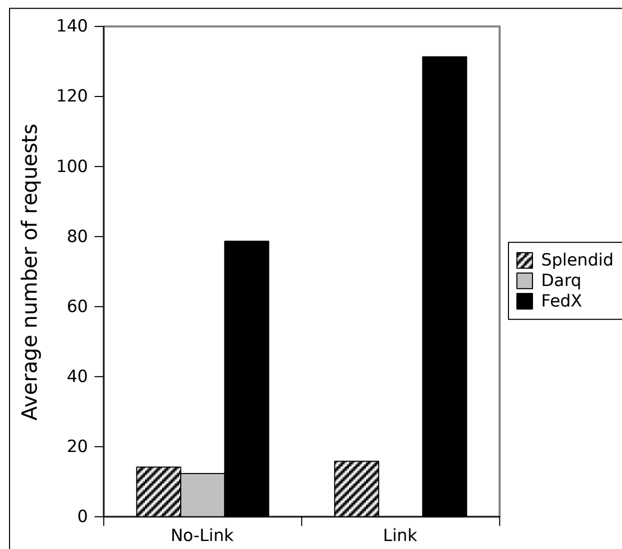


Figure 5.5: Average Number of Requests

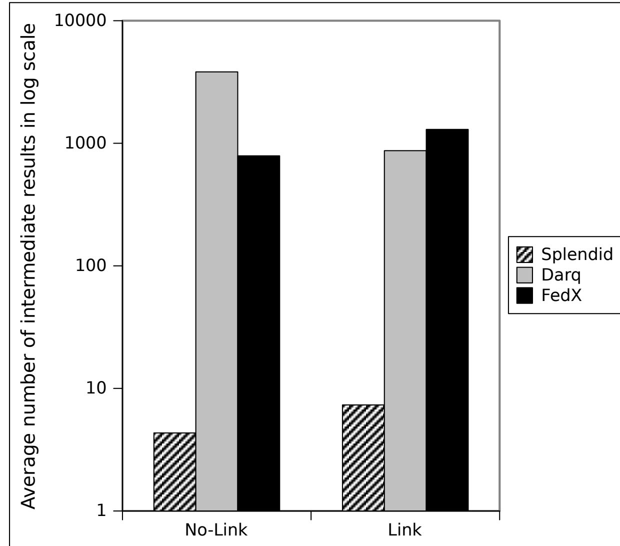


Figure 5.6: Average Intermediate Results (in log scale)

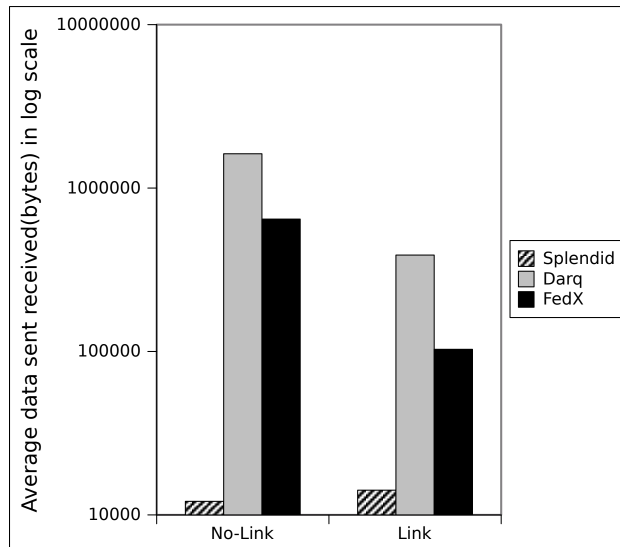


Figure 5.7: Average Data Transmission (in log scale)

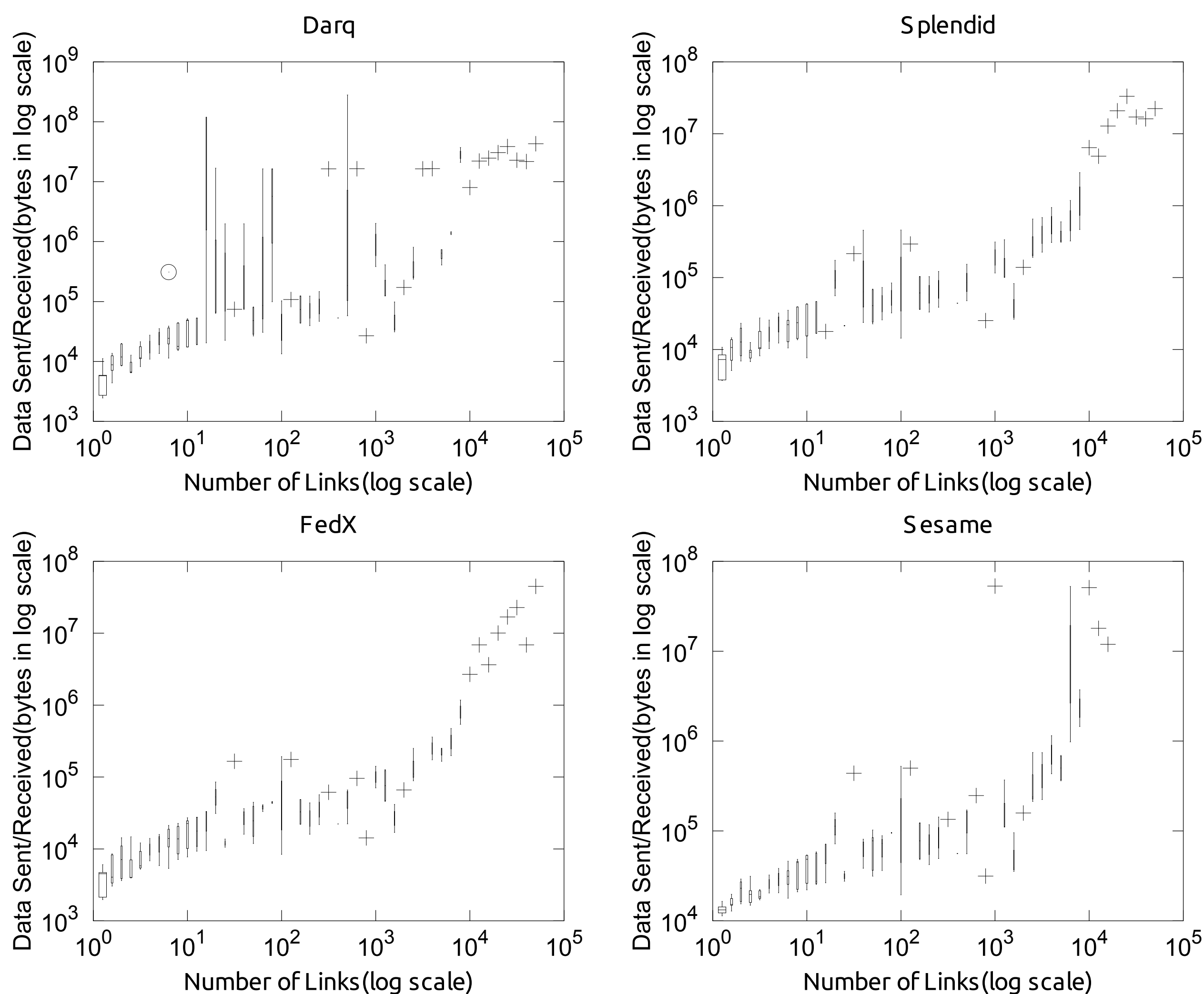


Figure 5.8: Average Data Sent and Received (Bytes) Versus the number of Links (log scale)

Query	1	2	3	4
#Correct Results	0%	90%	80%	100%

Table 5.2: The Accuracy of Query Results (Experiment Three)

engine and SPARQL endpoints. As indicated in Figure 5.8, the performance of Splendid, DARQ, and FedX first increases gradually and then starts rising sharply after 6000 links. Sesame transmits more data than the other federated engines. Thus, in the evaluation, Sesame time-outs occur for queries with more than 6000 links. In general, the federated engines generate higher data transmission as the number of links increases because the number of links is highly related to the intermediate results which eventually influences the amount of data transmission.

Results from the third experiment are summarised in Table 5.2. The first query aimed to map inactive-ingredients from two different sources, Dailymed and Drugbank. In this query, the inactive-ingredients of Dailymed are mapped to the drug names from Drugbank. Since the drug label in Drugbank mainly covers active-ingredients, this query failed to return the generic name of the drug. The second query mapped the active-ingredients from Dailymed to the drugs label from Drug Bank. This query has returned the correct results. Since both the active-ingredient and drug labels refer to the generic names of a drug, the second query retrieved correct results for all cases when a drug has only one active-ingredient. If a drug contains more than one active-ingredient (e.g *Zestoretic*), the query retrieves only one of the active-ingredients). The third query lists the drug names and their relation to generic names by using the *active-Moiety* label of Dailymed. This query returns the correct results with exceptions of *Vasocidin* and *Zestoretic* cases. In these cases, there is more than one active-ingredient, but that query is only able to retrieve one active-ingredient. In the fourth query, brand names are mapped using *genericDrug* link. This query returned the correct results. Since the links generation is generally validated in advance, the results of the query that exploits links is more accurate than the results of the query without using any link. The results of each query are detailed in Appendix D.

5.7 Conclusion

As part of the characteristics of a federated SPARQL query dataset, links have an important role in navigating from one dataset to other datasets, thus we investigate the impact of the existence of the links in the context of federated SPARQL querying. In order to observe the effects of the links, we run object similarity and using links query approaches over four federated engines: DARQ, Splendid, FedX and Sesame. Our experimental results reveal that the links can improve the performance of the federated engine if the federated engine applies the Nested-loop join strategy. Additionally, by using links, the number of query answers is higher than the query results without any link as a query without a link sometimes faces the case sensitivity and typo problems. Further, the accuracy of query results that uses the links is higher than the accuracy of query results without any link involved because links generation is normally curated by an expert.

Chapter 6

Dataset for Query Federation Benchmark*

This chapter explains several existing partition strategies that have been applied in a centralised RDF repository. We then adapt these strategies to divide a real dataset that will be used for benchmarking a federated SPARQL query. The results of these partitions are several datasets with different shapes such as number of triples, number of entities etc.

6.1 The states of the art of RDF data partitions

We first study data partition approaches for clustering a centralised RDF repository. To provide a better understanding, we use sample data in Listing 6.1.

6.1.1 METIS Partition

Often, with graph partitions, attempts are made to split the nodes and vertices uniformly in terms of the overall number. One notable graph partition for scalable data is METIS [Karypis and Kumar, 1998]. To split up the graph, METIS has three steps: graph coarsening, partitioning and un-coarsening. During the coarsening phase, the initial graph is converted to smaller graphs successively by removing edges between two vertices repeatedly. Once the smallest graph is created, the partition process will be applied. The partitioned coarsen graph will be the initial partition for bigger graph in the un-coarsening process. The aim

*Parts of this chapter have been published as [Rakhmawati and Hausenblas, 2012; Rakhmawati et al., 2014b]

Listing 6.1: Dailymed Sample Triples

```
dailymeddrug:82 a dailymed:drug .
dailymeddrug:82 dailymed:activeingredient dailymeding:Phenytoin .
dailymeddrug:82 rdfs:label "Dilantin-125 (Suspension)" .

dailymeddrug:201 a dailymed:drug .
dailymeddrug:201 dailymed:activeingredient dailymeding:Ethosuximide .
dailymeddrug:201 rdfs:label "Zarontin (Capsule)" .

dailymedorg:Parke-Davis a dailymed:organization .
dailymedorg:Parke-Davis rdfs:label "Parke-Davis" .
dailymedorg:Parke-Davis dailymed:producesDrug dailymeddrug:82 .
dailymedorg:Parke-Davis dailymed:producesDrug dailymeddrug:201 .

dailymeding:Phenytoin a dailymed:ingredients .
dailymeding:Phenytoin rdfs:label "Phenytoin" .

dailymeding:Ethosuximide a dailymed:ingredients .
dailymeding:Ethosuximide rdfs:label "Ethosuximide" .
```

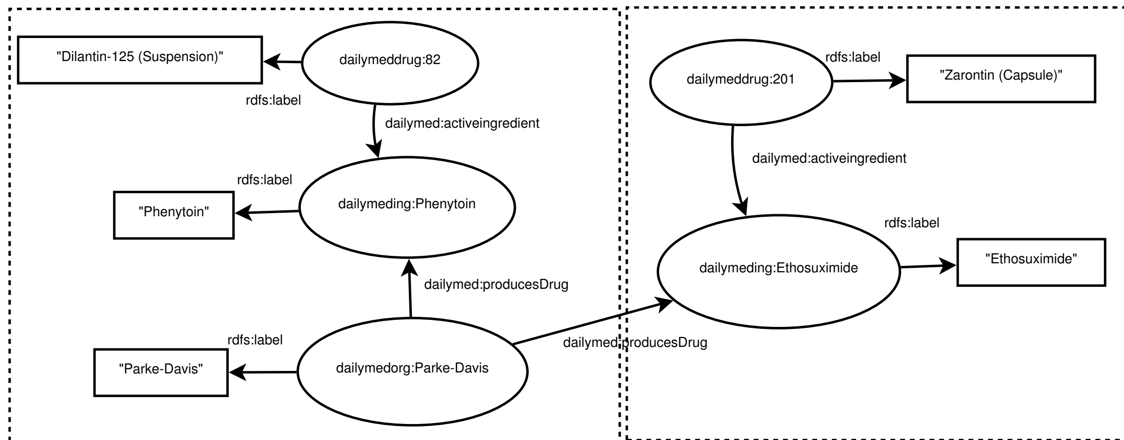


Figure 6.1: Example of METIS Partition applied

of this partition scheme is to reduce the communication needed between machines during the query execution process, by storing the connected components of the graph in the same machine. As shown in Figure 6.1, by using the METIS partition approach, there is only one link (`dailymed:producesDrug`) between two partitions; and the related triples are grouped in a single partition.

6.1.2 Vertically Partitioned Tables

This method, proposed by [Abadi et al., 2007], distributes triples based on its predicates. By excluding predicates, a table stores subjects and its associated object in the same row. Given a source graph in Listing 6.1, Vertically Partitioned Tables scheme generates four tables that can be found in Table 6.1.

rdf:type	
Subject	Object
dailymeddrug:82	dailymed:drug
dailymeddrug:201	dailymed:drug
dailymedorg:Parke-Davis	dailymed:organization
dailymeding:Phenytoin	dailymed:ingredients
dailymeding:Ethosuximide	dailymed:ingredients

rdfs:label	
Subject	**Object**
dailymeddrug:82	Dilantin-125 (Suspension)
dailymeddrug:201	Zarontin (Capsule)
dailymedorg:Parke-Davis	Parke-Davis
dailymeding:Phenytoin	Phenytoin
dailymeding:Ethosuximide	Ethosuximide

dailymed:activeingredient	
Subject	**Object**
dailymeddrug:82	dailymeding:Phenytoin
dailymeddrug:201	dailymeding:Ethosuximide

dailymed:producesDrug	
Subject	**Object**
dailymedorg:Parke-Davis	dailymeddrug:82
dailymedorg:Parke-Davis	dailymeddrug:201

Table 6.1: Example of Vertically Partitioned Tables applied

6.1.3 Property Tables

Wilkinson [Wilkinson, 2006] suggested a method for storing RDF data in a relational database, called Property Table (PT). There are two kinds of PT partitions: *Clustered Property Table*, in which RDF with the same predicate are grouped in a property table and *Property-class Table* in which each class is stored in a single table. The remaining triples which do not belong to the Property-Tables are stored to a *Left-Over-Triples* table.

The property table for a graph in Listing 6.1 has three columns: Subject, `rdfs:label` and `rdf:type` in which all entities in the graph are included in the Property-Table (Table 6.2). As shown in Table 6.3, when the Property-Class-Table is applied to the graph in Listing 6.1, three Property-Class-Tables are generated, but no Left-Over-Triples table is produced.

Property Table		
Subject	**rdfs:label**	**rdf:type**
dailymeddrug:82	Dilantin-125 (Suspension)	dailymed:drug
dailymeddrug:201	Zarontin (Capsule)	dailymed:drug
dailymedorg:Parke-Davis	Parke-Davis	dailymed:organization
dailymeding:Phenytoin	Phenytoin	dailymed:ingredients
dailymeding:Ethosuximide	Ethosuximide	dailymed:ingredients

Left-over Triples		
Subject	**Predicate**	**Object**
dailymedorg:Parke-Davis	dailymed:producesDrug	dailymeddrug:82
dailymedorg:Parke-Davis	dailymed:producesDrug	dailymeddrug:201
dailymeddrug:82	dailymed:activeingredient	dailymeding:Phenytoin
dailymeddrug:201	dailymed:activeingredient	dailymeding:Ethosuximide

Table 6.2: Example of Clustered Property Table applied

dailymed:drug		
Subject	**dailymed:activeingredient**	**rdfs:label**
dailymeddrug:82	dailymeding:Phenytoin	Dilantin-125 (Suspension)
dailymeddrug:201	dailymeding:Ethosuximide	Zarontin (Capsule)

dailymed:organization		
Subject	**rdfs:label**	**dailymed:producesDrug**
dailymedorg:Parke-Davis	Parke-Davis	dailymeddrug:82
dailymedorg:Parke-Davis	Parke-Davis	dailymeddrug:201

dailymed:ingredients	
Subject	**rdfs:label**
dailymeding:Phenytoin	Phenytoin
dailymeding:Ethosuximide	Ethosuximide

Table 6.3: Example of Property-class Table Applied

The fragmentation results of these partition approaches are stored in a relational database.

6.1.4 Class Partition

Class Partition divides a dataset based on its classes. This partition was used for DARQ evaluation [Quilitz and Leser, 2008].

6.1.5 Hash Partition

In this approach, each vertex is distributed randomly based on a hash function; as a consequence the data that is related to each other potentially ends up in different partitions. Eventually, this distribution could pose a problem in queries, since it induces more traffic between the federated engine and the data-source. The hash partition was proposed in SHARD [Rohloff and Schantz, 2010], which uses Hadoop[1] to provide the hashing function for each distinct subject.

6.1.6 Discussion

We have reviewed four state of the art partitions for clustering a centralised RDF repository. In order to minimize links between sources, we take METIS partition into account to divide a dataset. Due to the high numbers of properties in a dataset, we do not consider the Vertically Partitioned Tables to divide our dataset. The Property Tables stores the partition results in a relational database, while we only use an RDF store. However, we adopt these partitions to split the data with respect to its class information. We do not take the hash partition into account in our partition since a SHARD requires Hadoop for querying distributed data in a single SPARQL endpoint and therefore, it is not suitable for our set-up system.

6.2 Dataset Generation

Realistic data generation for benchmarking is a relevant research question in both the Semantic Web area and other mature fields like relational databases. As reported in Chapter 4, several existing evaluations divided a single dataset into smaller datasets. In a centralised RDF repository cluster, several machines need to communicate with each other in order to execute a query, whereas in a federated SPARQL query, there is no interaction amongst SPARQL endpoints. A federated engine has a role to communicate to each SPARQL endpoint during query execution in the federated SPARQL query. Although the machine communication of

[1] http://hadoop.org

centralised RDF repository cluster and the federated SPARQL query are different, we apply some of above centralised RDF repository cluster strategies to generate several synthetic datasets for benchmarking a federated SPARQL query. The following items are the essential requirements that need to be considered for the generation of datasets for assessing SPARQL query federation frameworks:

1. **Number of sources**, a federated SPARQL query framework consists of more than one source. Our work [Rakhmawati and Hausenblas, 2012] reported that the number of sources has a positive linear relationship with number of requests and response time.

2. **Distribution**, The content distribution of a dataset will influence how many sources are involved to answer a query. In terms of the RDF dataset, the content distribution of a dataset includes the triples distribution, entities distribution, properties distribution, and classes distribution. The distribution in this context means how triples, entities, properties and classes are spread out through the sources.

In this section we will present steps of partition approaches namely, METIS partition, entity partition, class partition, triples partition, property partition and hybrid partition. We will evaluate for each partition approach in Chapter 7.

6.2.1 METIS Partition

RDF Data can be represented as a graph. Inspired by a centralised RDF repository cluster [Huang et al., 2011], we performed a graph partition over our dataset by using METIS [Karypis and Kumar, 1998]. The METIS is considered for our partition tool because the METIS's output minimises the links between partitions.

1. We first identify a list of connections of subject and object in different triples. We only consider a URI object which is also a subject in other triples. Intuitively, the reason is that an object which appears as the subject in other triples can create a connection if the triples are located in different dataset partitions. $V(D)$ denotes a set of pairs of subject and object that form a path (see path definition in Definition 5.1) in the dataset D.

2. Assign a numeric identifier for each pair of subject and object $V(D)$

3. Create a list of sequential adjacent vertexes for each vertex then uses it as an input of the METIS API.

4. Run METIS to divide the vertexes. METIS produces a list of the partition number of the vertexes.

5. Distribute each triple based on the partition number of its subject and object.

Consider an example, given a dataset in Listing 6.1, then

$$V(D) = \{(\text{dailymeddrug:82}, \text{dailymeding:Phenytoin}), (\text{dailymeddrug:201}, \text{dailymeding:Ethosuximide}), (\text{dailymedorg:Parke-Davis}, \text{dailymeddrug:82}), (\text{dailymedorg:Parke-Davis}, \text{dailymeddrug:201})\}$$

An identifier value starts from one, and we then increase the identifier value by one. For example, we set the identifier of each entity as follows:

- `dailymeddrug:82`=1
- `dailymeding:Phenytoin`=2
- `dailymeddrug:201`=3
- `dailymeding:Ethosuximide`=4
- `dailymedorg:Parke-Davis`=5.

After that, we can create a list of sequential adjacent vertexes $V(D)$: $\{(2,5), 1, (4,5), 3, (1,3)\}$. For instance, we divide the sample of the dataset into 2 partitions, then output of the METIS partition is $\{1,1,2,2,1\}$ where each value is the partition number for each vertex. Based on the METIS output, we can say that `dailymeddrug:82` belongs to partition 1, `dailymeding:Phenytoin` belongs to partition 1, `dailymeddrug:201` belongs to partition 2 and so on. In the end, we have the two following partitions:

Partition 1: all triples that contain `dailymeddrug:82`, `dailymeding:Phenytoin` and `dailymedorg:Parke-Davis`
Partition 2: all triples that contain `dailymeddrug:201` and `dailymeding:Ethosuximide`

The result of the METIS partition is also shown in Figure 6.1.

6.2.2 Entity Partition

To begin with, we define an entity and a class as follows:

Definition 6.1 *if there is a triple $t(s, p, o)$ and $p =$ rdf:type, then s is an entity and o is a class of entity s.*

The goal of entity partition is to distribute number of entities evenly in each partition. Different classes can be located in a single partition. However, entities of the same class should be grouped in the same partition until the number of entities reaches the maximum number of entities for each source. We initially create a list of subjects along with its class ($E(D)$) as follows:

Definition 6.2 *The set $E(D)$ of pairs of the entity and its class in dataset D is*
$E(D) = \{(s, o) | \exists (s, \textit{rdf:type}, o) \in D\}$

To do entity partition, we deploy the following methods:

1. Sort $E(D)$ by its class o .

2. Store each pair of the subject and object in a single partition until the number of pairs of the subject and object equals to the ratio of the total number of pairs of subject and object and the number of partitions.

3. Distribute the remainder of the triples in the dataset based on the subject location.

For instance, given a dataset in Listing 6.1, then

$$E(D) = \{(\text{dailymeddrug:82}, \text{dailymed:drug}), (\text{dailymeddrug:201}, \text{dailymed:drug}), (\text{dailymedorg:Parke-Davis}, \text{dailymed:organization}), (\text{dailymeding:Phenytoin}, \text{dailymed:ingredients}), (\text{dailymeding:Ethosuximide}, \text{dailymed:ingredients})\}$$

Suppose we split the dataset into two partitions, then the maximum number of entities for each partition is $\frac{|E(D)|}{number of partitions} = \frac{5}{2} = 3$ (ceiling of 2.5). We put `dailymeddrug:82`, `dailymeddrug:201` and `dailymedorg:Parke-Davis` in partition one and store the remainders of entities in partition two. As the final step, we distribute the related triples based on its subject partition number.

6.2.3 Class Partition

First, we distributed entities based on its class. After that, the related triples that belong to an entity are placed in the same machine. The class partition steps are explained as follows:

1. To begin with, we also create $E(D)$ which has been defined in Definition 6.2.

2. Distribute each triple based on the class of the subject.

Like entity partition example, we do the same step to generate $E(D)$. For instance, given $E(D$ as defined in Section 6.2.2, then we divide the datasets into three partitions as follows:

dailymed:drug	dailymed:organization	dailymed:ingredients
dailymeddrug:82	dailymedorg:Parke-Davis	dailymeding:Phenytoin
dailymeddrug:201	-	dailymeding:Ethosuximide

6.2.4 Property Partition

In our property partition, we do not have a Property-class table because we treat all properties in the same manner. We store the triples that have the same property to a single data-source. Due to a high number of properties in the dataset, more than one property may be stored in the same partition as long as we get similar numbers of triples among the partitions. In summary, our query set partition steps are described as follows:

1. Group the triples based on its property.

2. Store each group in a partition until the number of partition triples is less than or equal to the number of dataset triples divided by the number of partitions.

For instance, given a dataset as shown in Listing 6.1, then we have four properties: `rdf:type`, `dailymed:activeingredient`, `rdf:label` and `dailymed:producesDrug`. Suppose that we want to divide the dataset into two partitions, then the maximum number of triples in each partition is $\frac{thenumberoftriples}{thenumberofpartitions} = \frac{14}{2} = 7$. As the following step, we store the triples based on its property as follows:
Partition 1: five triples with `rdf:type` property, two triples with `dailymed:activeingredient` property
Partition 2: five triples with `rdfs:label` property, two triples with `dailymed:producesDrug`

6.2.5 Triples Partition

Performance of framework for federated SPARQL query is influenced not only by the federated engine solely, but also depends on the SPARQL endpoints within the federation framework. In order to keep balanced workload for SPARQL endpoints, we split up the triples of each source evenly because LUBM [Guo et al., 2005] mentioned that the number of triples can influence the performance of a RDF repository. If the triples partition applies over a given dataset in Listing 6.1 to divide the dataset into two partition, then partition 1 contains the following triples:

```
dailymeddrug:82 a dailymed:drug .
dailymeddrug:82 dailymed:activeingredient dailymeding:Phenytoin .
dailymeddrug:82 rdfs:label "Dilantin-125 (Suspension)" .
dailymeddrug:201 a dailymed:drug .
dailymeddrug:201 dailymed:activeingredient dailymeding:Ethosuximide .
dailymeddrug:201 rdfs:label "Zarontin (Capsule)" .
dailymedorg:Parke-Davis a dailymed:organization .
```

partition 2 contains the following triples:

```
dailymedorg:Parke-Davis rdfs:label "Parke-Davis" .
dailymedorg:Parke-Davis dailymed:producesDrug dailymeddrug:82 .
dailymedorg:Parke-Davis dailymed:producesDrug dailymeddrug:201 .
dailymeding:Phenytoin a dailymed:ingredients .
dailymeding:Phenytoin rdfs:label "Phenytoin" .
dailymeding:Ethosuximide a dailymed:ingredients .
dailymeding:Ethosuximide rdfs:label "Ethosuximide" .
```

6.2.6 Hybrid Partition

The Hybrid Partition is a partitioning method that combines two or more previous partition strategies. For instance, if the number of triples that are related to a specific class is more than the number of triples in a partition, we can distribute the triples to another partition to equalise the number of triples. As shown in Listing 6.1, the number of triples that are related to `dailymed:drug` is six, while other classes have four triples. We can move one triple from `dailymed:drug` partition to other partitions.

6.3 DFedQ

We provide DFedQ, a tool for generating the following datasets that can be found at `https://github.com/nurainir/DFedQ`. This tool produces several datasets with different shapes such as the number of entities, the number of classes, the number of properties for each partition.

Chapter 7

On Metrics For Measuring Fragmentation Of Federation over SPARQL Endpoints*

In this chapter we investigate the effect of data distribution on the federated engine performance. We propose two composite metrics to calculate the presence of classes and properties across sources. These metrics can provide an insight into the data distribution in the dataset which ultimately can determine the communication cost between the federated engine and SPARQL endpoints.

7.1 Motivating Example

Processing a federated query in Linked Data is challenging because we need to consider the number of the sources, the source locations and heterogeneous systems such as the hardware, the software, the data structure and the distribution. A federated SPARQL query can be formulated by using the `SERVICE` keyword. Nevertheless, determining the data source address that follows `SERVICE` keywords can be an obstacle in writing a query because a priori knowledge data is required. To address this issue, several approaches described in Chapter 3 have been developed with the objective of hiding the `SERVICE` keyword and the source locations from the user. In these approaches, the federated engine receives a query from the user, parses the query into sub queries, decides the location of each sub query and distributes the sub queries to the relevant sources. A sub query can be delivered to more

*Parts of this chapter have been published as [Rakhmawati et al., 2014b]

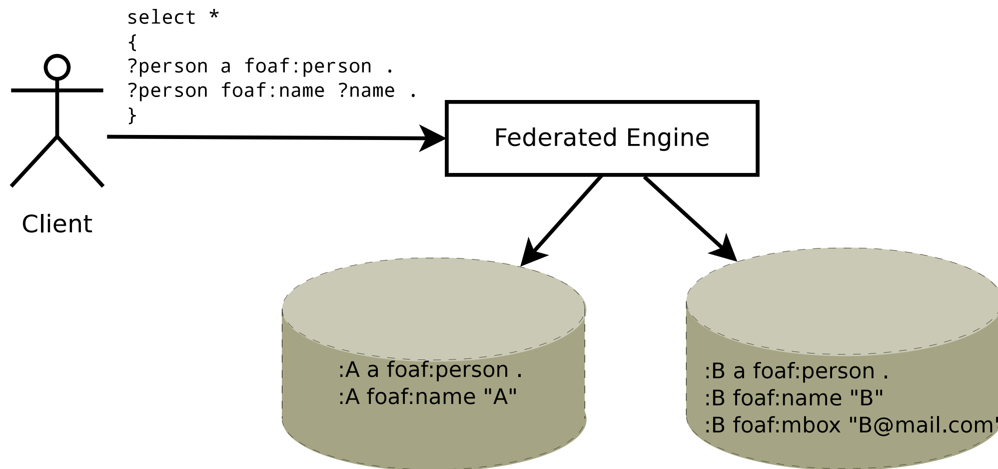


Figure 7.1: Example of a federated SPARQL query involving many datasets

than one data-source if the desired answer occurs in multiple sources. Thus, the distribution of the data can affect the federation performance [Rakhmawati and Hausenblas, 2012]. As an example, consider two sources shown in Figure 7.1. Each source contains a list of personal information using the FOAF[1] vocabulary. If the user asks for the list of all person names, the federated engine must send a query to all sources. Consequently, the communication cost between the federated engine and data-sources would be expensive.

Federated engines typically use a data catalogue to predict the most relevant sources for a subquery. The data catalogue generally consists of a list of predicates and classes (Chapter 3.2.4). Apart from deciding the destination of the sub queries, a data catalogue can assist the federated engine to generate a set of query execution plans. Hence, we analyse the distribution of classes and properties throughout the dataset. The dataset used in this work is defined in Definition 3.1. First of all, we determine a list of properties and classes in the source-set as follows:

Definition 7.1 *Suppose d is a source in the source-set D, then the set $P(d)$ of properties in the source d is defined as $P(d) = \{p|\exists(s, p, o) \in d \wedge d \in D\}$ and the set $P(D)$ of properties in the source-set D is defined as $P(D) = \{p|p \in P(d) \wedge d \in D\}$*

We also consider the occurrences of classes to cover a SPARQL query that contains `rdf:type` following by the class name.

Definition 7.2 *Suppose d is a source in the source-set D, then the set $C(d)$ of classes in the source d is defined as $C(d) = \{c|\exists(s, rdf{:}type, c) \in d \wedge d \in D\}$ and the set of classes in*

[1] `http://xmlns.com/foaf/spec/`

the source-set D is defined as $C(D) = \{c | c \in C(d) \land d \in D\}$

Given source-set $D = \{d_1, d_2\}$ as shown in Figure 7.1, then $P(d_1) = \{$rdf:type,foaf:name$\}$, $P(d_2) = P(D) = \{$rdf:type,foaf:name, foaf:mbox$\}$ and $C(d_1) = C(d_2) = C(D) = \{$foaf:person$\}$.

7.1.1 Spreading Factor of a Dataset

With the above definitions of class and property, we can describe the calculation of the spreading factor. The spreading factor of a dataset is based on the occurrence of classes and properties in a source-set. In this calculation, we do not count the number of times a class and property that are found in the source d because the federated engine usually relies on the presence of properties in order to predict which SPARQL endpoints that can answer a sub-query (see Section 4.5 Source Selection). Given a source-set D that contains a set of sources d, the normalisation of number of occurrences of properties in the source-set D ($OCP(D)$)is calculated as follows:

$$OCP(D) = \frac{|P(D)|}{\sum_{d \in D} |P(d)|} \tag{7.1}$$

Given the source-set as shown in Figure 7.1, we can calculate $OCP(D) = \frac{3}{2+3} = 0.6$. However, Equation 7.1 cannot reflect the distribution of properties in the source set. For instance, given four distinct properties p_1, p_2, p_3, p_4 that are distributed to two distinct sources d_1, d_2. Suppose that we have two source sets D_1 and D_2 as follows:

$D_1 = \{d_1, d_2\}$ where $P(d_1) = \{p_1, p_2\}$ and $P(d_2) = \{p_3, p_4\}$

$D_2 = \{d_1, d_2\}$ where $P(d_1) = \{p_1, p_2, p_3\}$ and $P(d_2) = \{p_4\}$

D_1 and D_2 have different distributions of properties with the same number of distinct properties, but have the same OCP value as follows:

$OCP(D_1) = \frac{4}{2+2} = 1$

$OCP(D_2) = \frac{4}{3+1} = 1$

The OCP should produce the same value for the same distribution, therefore we modify Equation 7.1 as follows:

$$OCP(D) = \frac{\sum_{d \in D} |P(d)|}{|P(D)| \times |D|} \tag{7.2}$$

To provide a clarity, we generate various combinations of distribution properties in Table 7.1 where the number of sources is 2 and the number of distinct properties ($P(D)$) are 3, 4

$\|P(D)\|$	$\|P(d_1)\|$	$\|P(d_2)\|$	OCP_1	OCP_2
3	1	2	1	0.5
3	1	3	0.75	0.67
3	2	3	0.6	0.83
3	2	2	0.75	0.67
3	3	3	0.5	1
4	1	3	1	0.5
4	1	4	0.8	0.625
4	2	2	1	0.5
4	2	3	0.8	0.625
4	2	4	0.67	0.75
4	3	3	0.67	0.75
4	3	4	0.57	0.875
4	4	4	0.5	1
5	1	4	1	0.5
5	1	5	0.83	0.6
5	2	3	1	0.5
5	2	4	0.83	0.6
5	2	5	0.71	0.7
5	3	3	0.83	0.6
5	3	4	0.71	0.7
5	3	5	0.625	0.8
5	4	4	0.625	0.8
5	4	5	0.56	0.9
5	5	5	0.5	1

Table 7.1: The combinations of distribution of properties

and 5 properties. OCP_1 is the value of OCP for Equation 7.1, while OCP_2 is the value of OCP for Equation 7.2. As shown in Table 7.1, the value of OCP_1 is 1 for more than one distribution with the same number of $P(D)$, while the value of OCP_2 is 1 when all properties occurs in all sources. Further, the value of OCP_2 increases when the number of properties in each sources increases (Figure 7.3), while the value of OCP_2 is inconsistent (Figure 7.2).

The normalisation of number of occurrences of classes in the source-set D ($OCC(D)$) also adopt Equation 7.2.

$$OCC(D) = \frac{\sum_{d \in D} |C(d)|}{|C(D)| \times |D|} \tag{7.3}$$

$OCP(D)$ and $OCC(D)$ have a range value from 0 to 1. If all properties occur in all datasets, Equation 7.2 is designed to result in the value 1. Likewise, if all classes occur in all datasets,

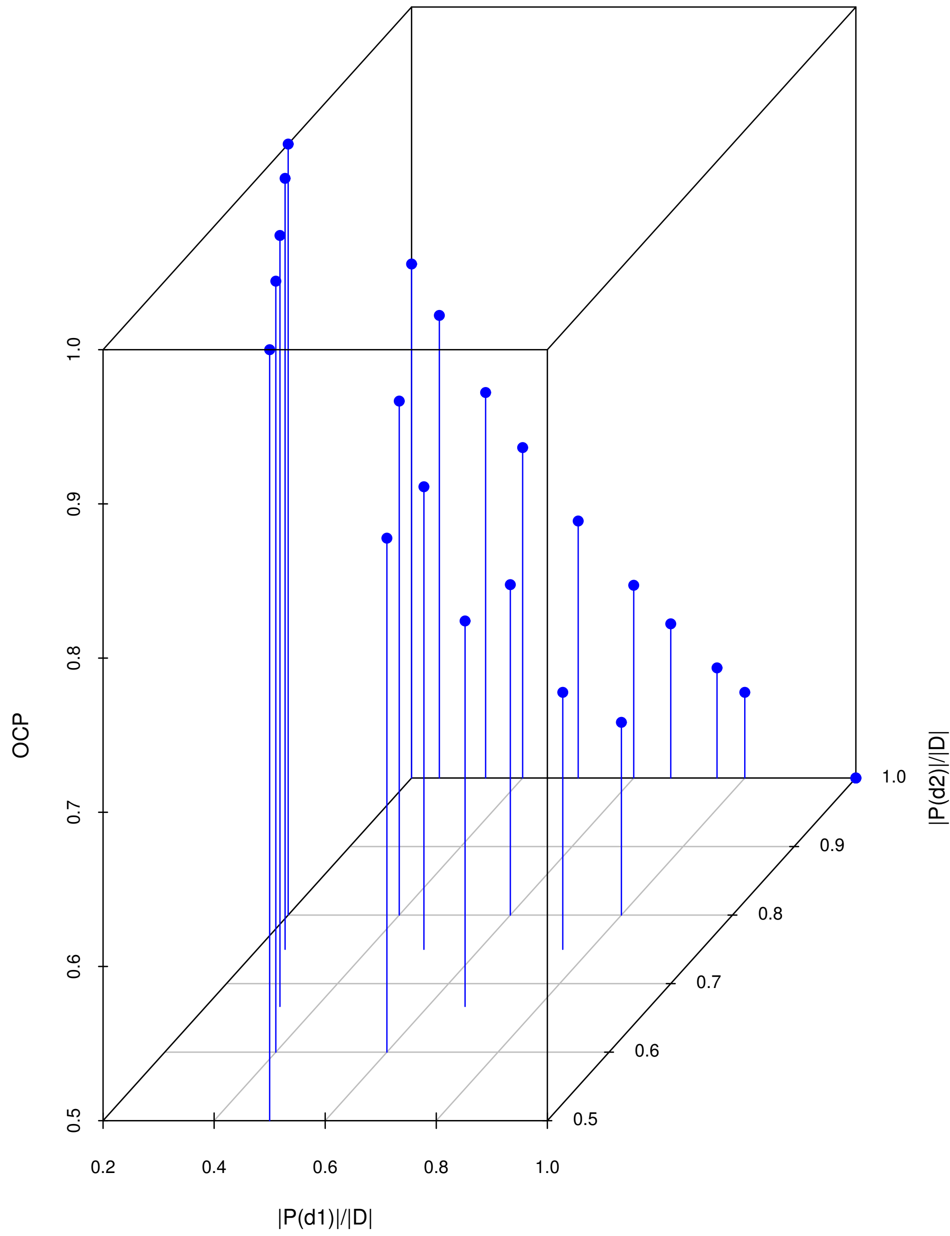


Figure 7.2: The OCP values for Equation 7.1

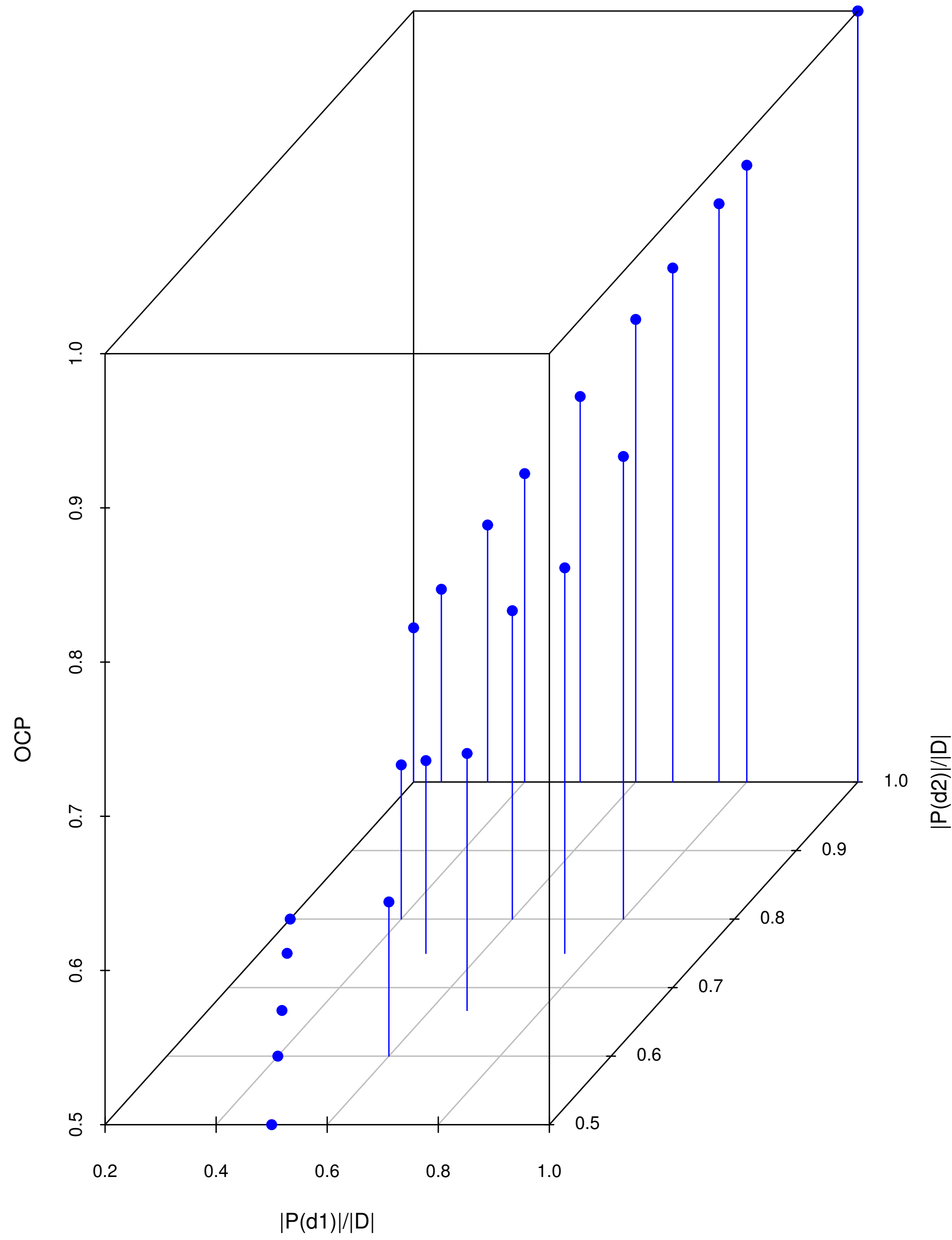


Figure 7.3: The OCP values for Equation 7.2

Equation 7.3 is designed to result in the value 1.

Inspired by the *F-Measure* function [van Rijsbergen, 1979], we combine $OCP(D)$ and $OCC(D)$ into a single metric which is called the spreading factor $\Gamma(D)$ of the dataset D. We choose the F-Measure because it is a weight function that combines two different metrics. Further, we can set a weight to give different emphasis to classes and properties.

$$\Gamma(D) = \frac{(1+\beta^2)OCP(D) \times OCC(D)}{\beta^2 \times OCP(D) + OCC(D)} \tag{7.4}$$

β is an arbitrary value, we can assign any value for β. The high Γ value indicates that the class and properties are spread out over the dataset.

Using our previous example in which we define $P(d_1)$, $P(d_2)$, $P(D)$, $C(D)$, $C(d_1)$,$C(d_2)$, then we can calculate $OCP(D) = \frac{2+3}{3X2} = 0.833$ and $OCC(D) = \frac{1+1}{1X2} = 1$. Given $\beta = 0.5$, we obtain $\Gamma(D) = 0.961$

7.1.2 Spreading Factor of a Dataset associated with the Query set

The spreading factor of a dataset reveals how the classes and properties are distributed over the dataset. However, a query may not consist of all properties and classes in the dataset. Thus, it is necessary to quantify the spreading factor of the dataset with respect to the query set. For instance, a user delivers the query that consists of rdf:type and foaf:name (Figure 7.1). The federated engine will not seek the location of foaf:mbox because foaf:mbox are not included in the query. Therefore, we modify $OCP(D)$ (Equation 7.2) and $OCC(D)$ (Equation 7.3). These equations will only cover particular classes and properties that are included in the query.

$$OCP(p, D) = \frac{|\bigcup_{\forall d \in D}\{d|p \in P(d)\}|}{|D|} \tag{7.5}$$

$$OCC(o, D) = \frac{|\bigcup_{\forall d \in D}\{d|o \in C(d)\}|}{|D|} \tag{7.6}$$

Given a query set $Q = \{q_1, q_2, \cdots, q_n\}$, the Q-spreading factor γ of the dataset D associated with query set Q is computed as

$$\gamma(Q, D) = \sum_{\forall q \in Q} \frac{\sum_{\forall \tau \in q} OC(\tau, D)}{|Q|} \tag{7.7}$$

where the occurrences of the class and property for τ is specified as

$$OC(\tau, D) = \begin{cases} OCC(o_\tau, D) & \text{if } p_\tau \text{ is rdf:type} \\ & \quad \wedge o_\tau \notin V \\ OCP(o_\tau, D) & \text{if } o_\tau \text{ is not rdf:type} \\ & \quad \wedge p_\tau \notin V \\ \frac{\sum_{\forall d \in D} |P(d)|}{|D|} & \text{otherwise} \end{cases}$$

Consider an example, given a query and a source-set as shown in Figure 7.1, then $OC($?person a `foaf:person` $, D) = 1$ and $OC($?person foaf:name ?name $, D) = 1$ because foaf:person and foaf:name are located in two different sources. As a result, the q-spreading factor $\gamma(Q, D)$ is $\frac{1+1}{1} = 2$

7.2 Evaluation

The objective of our evaluation is to show that our spreading factor metrics are highly related to the communication cost between a federated engine and SPARQL endpoints. We compute the correlation between the communication cost and our spreading factor metrics.

We perform the following evaluation steps:

- In order to evaluate our metrics, we use partition strategies to generate different shapes of data distribution as described in Chapter 6. In total, we generate nine datasets by partitioning the native Dailymed dataset into three parts.
- After that, we design a static query set that will be executed over those data distributions.
- We only perform our observation on two federation over SPARQL endpoints systems, namely SPLENDID and DARQ. However, our work could also be implemented in other distributed query processing systems such as Hybrid-processing approaches [Umbrich et al., 2012c], link traversal approaches [Hartig, 2011] and the federation over single RDF repositories [Haase et al., 2010]. Query with a SERVICE keyword is also out of the scope of our study because this query only goes to the specified sources. In other words, the data distribution does not influence the performance of the federation engine while executing this query.

- We measure the bandwidth usage between the federated engine and SPARQL endpoints during runtime. Later on, this bandwidth usage is called as the volume of data transmission.
- We calculate the correlation between the spreading factor and the volume of data transmission by using the Pearson correlation.
- In the final step, we generate different spreading factor of the dataset by using different β values.

The details of our evaluation can be explained as follows:

7.2.1 System

We ran our evaluation on an Intel Xeon CPU X5650, 2.67GHz server with Ubuntu Linux 64-bit installed as the Operating System and Fuseki 1.0 as the SPARQL Endpoint server. For each dataset, we set up Fuseki on different ports. Each query was executed three times on two federation engines, namely SPLENDID and DARQ. These engines were chosen because SPLENDID employs VoID as a data catalogue that contains a list of predicates and entities while DARQ has a list of predicates which is stored in the Service Description. Apart from using VoID, SPLENDID also sends a SPARQL ASK query to determine whether or not the source can potentially return the answer. We explain the details of our dataset generation and metrics as follows:

7.2.2 Query set

In order to evaluate the dataset, we design a query set(Table 7.2) comprising 16 queries.These 16 queries include all classes and properties in Dailymed dataset. We prefer seeing the coverage of a query over partitions, therefore the operator and modifier are not included in our queries. To reflect real word queries, we design the queries based on the distribution of properties and classes, selectivity, and the shape of the path query. The distribution of properties and classes can be computed as the spreading factor metric. We use unpopular and popular properties. In terms of selectivity, we use variables in various positions. For instance, queries 3 and 4 could cover multiple entities since their predicate is a variable. Although the predicate is not a variable, query 5 also involves multiple entities since it consists of an unbound subject, `rdfs:label` as the predicate and an unbound object that could be located in every partition. There are three shapes of the query path, namely chain,

QID	Across Entities	Star(S)/Chain(C)/Hybrid
1	No	S
2	No	S
3	No	-
4	Yes	-
5	Yes	-
6	No	S
7	No	S
8	No	S
9	No	S
10	No	S
11	Yes	H
12	Yes	H
13	Yes	H
14	Yes	C
15	Yes	H
16	Yes	H

Table 7.2: Query Characteristic

star and hybrid. A '`-`' refers to a single path query. We limited the query processing duration to one hour. The query details can be found in Appendix B.1.

7.2.3 Dataset

9 partitions were generated as listed in Table 7.3. Three triple partition datasets (TD, $TD2$, $TD3$) were created. TD is obtained by partitioning the native Dailymed dataset into three parts. $TD2$ and $TD3$ are generated by picking a random starting point within the Dailymed dump file(by picking a random line number). Since the number of triples in each dataset of the Class Distribution CD are not equal, HD is created to distribute the triples evenly. However, the `rdf:type` property and the `rdfs:label` property are distributed evenly through all partitions in dataset $HD2$. This distribution is intended for balancing the workload amongst SPARQL endpoints since those properties are commonly used in our query set.

Figure 7.4 shows the calculation of predicates and classes occurrences. The spreading factor of datasets are shown in Figure 7.5. We assign $\beta = 2$ in order to put two times more emphasis on the classes than the properties because more than 50% of our queries contains `rdf:type`. As shown in these figures, the classes and properties are distributed over most

Dataset	Approach
GD	Graph Partition
CD	Class Partition
TD	Triples Partition
TD2	Triples Partition
TD3	Triples Partition
PD	Property Partition
ED	Entity Partition
HD	Entity, Class and Triple Partition
HD2	Entity Partition, `rdf:type` and `rdfs:label` distributed evenly

Table 7.3: Dataset used in evaluation

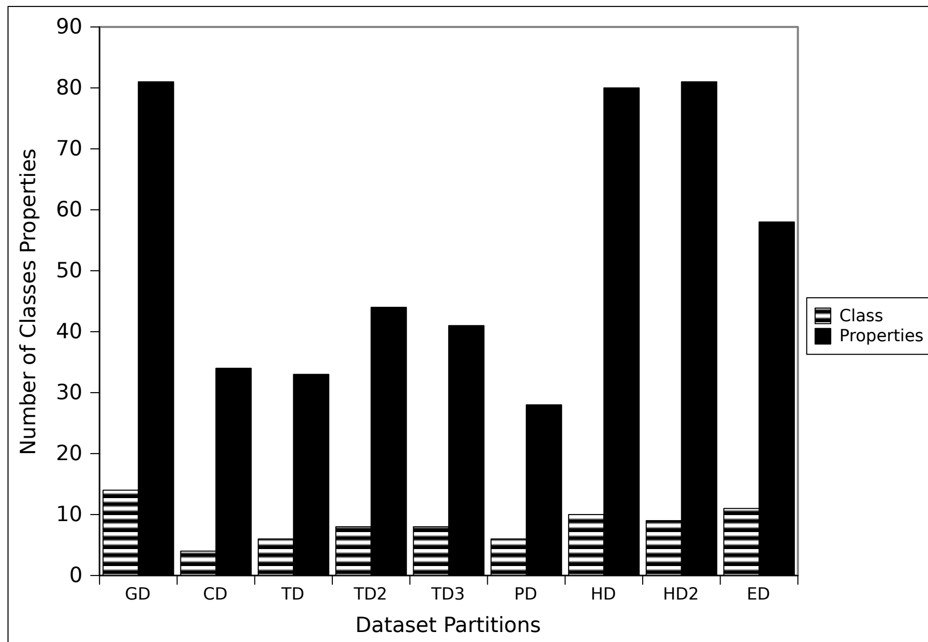


Figure 7.4: The occurrences of classes and properties in the dataset

of the partitions in the *GD* dataset. The *PD* has the lowest spreading factor among the datasets because each property occurs exactly in one and only one partition that contains `rdf:type`. The distribution of triples and coherence values among the datasets are presented in Figure 7.6 and 7.7 respectively. The dataset generation code and the generation results can be found at DFedQ github[2]

[2] `https://github.com/nurainir/DFedQ`

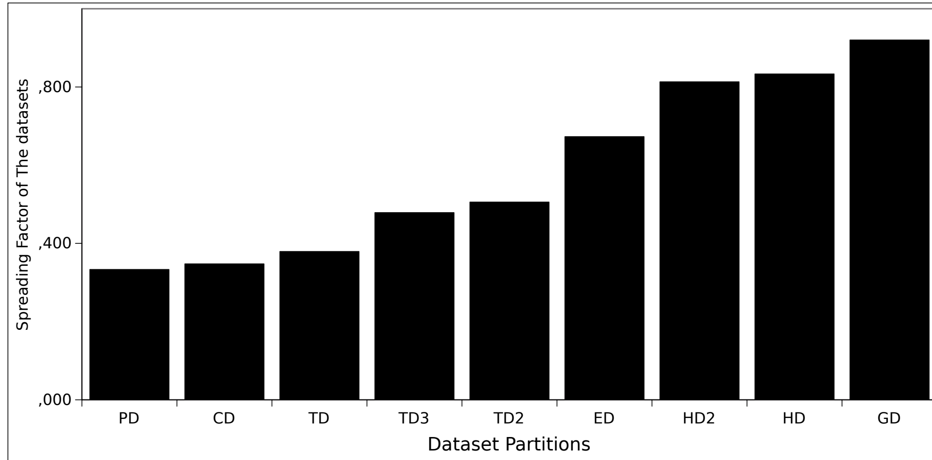


Figure 7.5: Spreading factor of the dataset

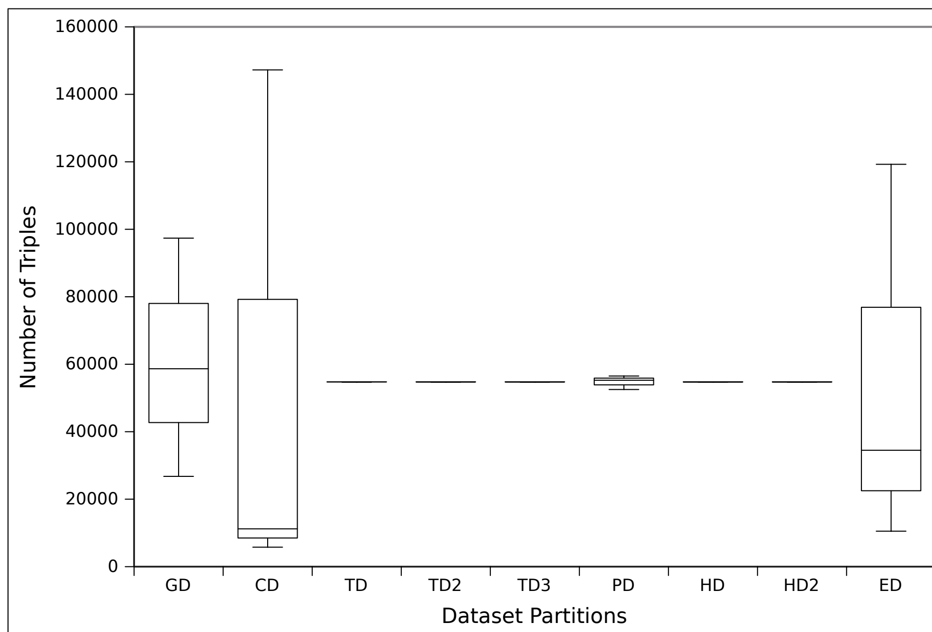


Figure 7.6: The distribution of number of triples amongst the dataset partitions

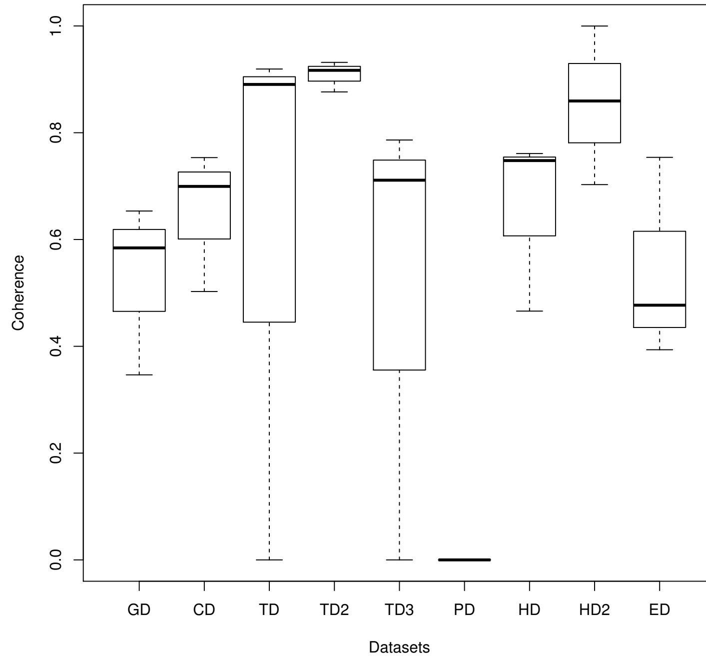


Figure 7.7: The distribution of coherence values amongst the dataset partitions

7.2.4 Metrics

To calculate the communication cost of the federated SPARQL query, the volume data transmission between the federated engine and SPARQL Endpoints is computed. The volume data transmission includes the amount of data both sent and received by the mediator. The final metrics shown in the following graphs is a composite metric as discussed in Chapter 4.

7.3 Results and discussion

In our previous experiment [Rakhmawati et al., 2014b], we set $\beta = 0.5$. As seen in Table 7.5, we achieve 72% correlation value. We then generate other spreading factors of the dataset by using $\beta = 0.6, 0.75, 1, 1.2, 1.5, 2$ and 2.5. Graph 7.8 shows that the correlation value increases linearly with β value from 0.5 to 2. Hence, $\beta = 2$ is the best value for our dataset. The β value is dependent to the dataset and the query set.

As seen in Figures 7.9- 7.10 and Table 7.4, the volume data transmission between DARQ and SPARQL endpoints is higher than the data transmission between SPLENDID and SPARQL endpoints. This is because DARQ never sends SPARQL ASK queries in order

Dataset	$\Gamma(D), \beta = 2$	$\gamma(Q, D)$	Splendid	DARQ
PD	0,33	0,81	1.108.667,53	1.400.185,82
CD	0,24	1,07	1.624.941,31	1.550.632,54
TD	0,34	1,33	1.516.277,90	1.712.768,39
TD3	0,45	1,36	1.787.506,38	1.958.427,22
TD2	0,46	1,64	1.713.565,46	1.784.818,20
ED	0,63	1,76	1.704.258,63	1.952.598,75
HD2	0,55	2,02	1.717.721,47	1.953.069,80
HD	0,61	2,19	1.734.167,18	1.953.348,40
GD	0,81	2,43	2.519.135,09	3.618.034,68

Table 7.4: Evaluation Results

	Splendid	DARQ
$\Gamma(D), \beta = 0.5$	0,72	0,73
$\Gamma(D), \beta = 2$	0,80	0,82
$\gamma(Q, D)$	0,73	0,76

Table 7.5: Pearson Correlation Evaluation Results

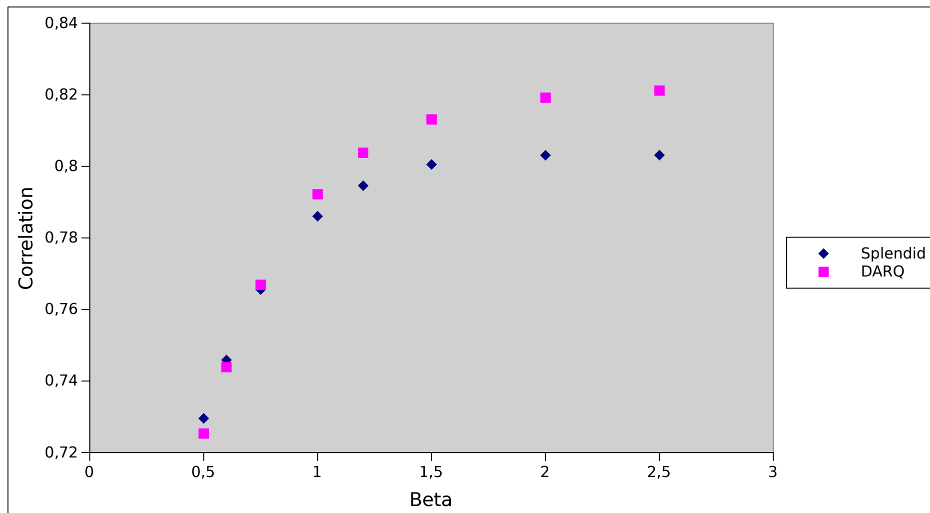


Figure 7.8: Pearson Correlation Vs β

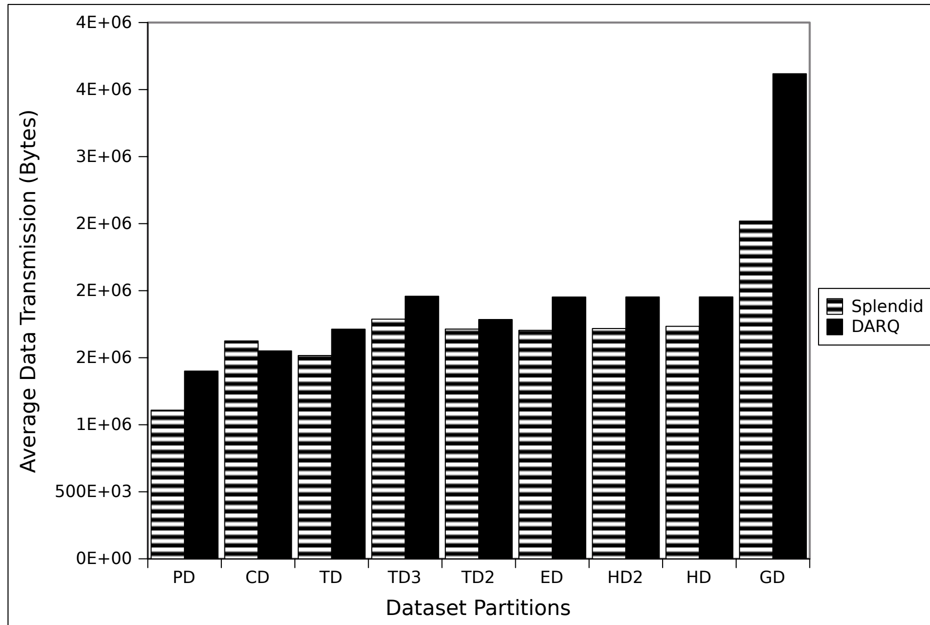


Figure 7.9: Average Data Transfer Volume Vs the spreading factor of Datasets (order by the spreading factor value)

to predict the most relevant source for each subquery. Table 7.5 shows a high positive correlation between data transmission and spreading factor metrics. It implies that there is a fairly strong relationship between data transmission and spreading factor metrics.

Overall,the data transmission increases gradually in line with the spreading factor of a dataset. However, the data transmission rises dramatically for *GD* distribution. This indicates that in the context of federated SPARQL queries, data clustering based on its property and class is better than data clustering based on related entities such as Graph Partition. The reason for this is that the source selection in federated query engine depends on classes and properties occurrences. Furthermore, when the federated engines generate query plans, they use an optimisation technique based on the statistical predicates and classes (Chapter 3).

7.4 Conclusion

We have implemented various data distribution strategies to partition classes and properties over dataset partitions in the previous chapter. We introduced two notions of dataset metrics, namely the spreading factor of a dataset and the spreading factor of a dataset associated with the query set. These metrics expose the distribution of classes and properties over the

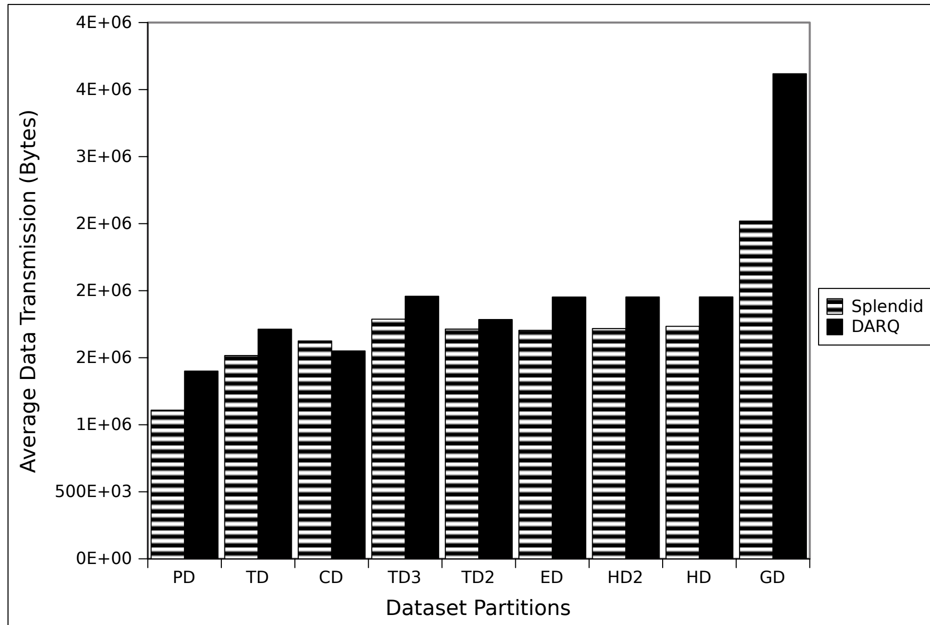


Figure 7.10: Average Data Transfer Volume Vs the Q-spreading factor of Datasets associated with the Queryset(order by the spreading factor value)

dataset partitions. Our experiment results revealed that the class and property distribution affects the communication cost between the federated engine and SPARQL endpoints. Partitioning triples based on properties and classes can minimize communication cost. However, such partitioning can also reduce the performance of SPARQL endpoints within the federation infrastructure. Further, it can also influence the overall performance of the federation framework.

Chapter 8

QFed: Query Benchmark Generator Based on Metrics and Characteristics of a dataset [*]

This thesis particularly observes three main components of benchmark: metric, dataset and query as shown in Figure 1.2. In the previous chapters, we have presented metrics (Chapter 4) for evaluating a federated SPARQL queries framework. We then concentrate on the dataset component in Chapter 5, 6 and 7. In this chapter, based on our findings in the previous chapter, we propose QFed for generating queries for benchmarking federated engines.

Most of the existing benchmark systems rely on a set of predefined static queries over a particular set of data sources. Such benchmarks are useful for comparing general purpose SPARQL query federation systems such as FedX, SPLENDID etc. However, special purpose federation systems such as TopFed, SAFE etc. cannot be tested with these static benchmarks since these systems only operate on the specific data sets and the corresponding queries. To facilitate the process of benchmarking for such special purpose SPARQL query federation systems, in this chapter we propose *QFed*, a dynamic query set generator for federated SPARQL query benchmarks. QFed takes into account the key characteristics of the dataset as well as SPARQL queries which has a direct impact on the performance evaluation of federated engines. *QFed* considers SPARQL query characteristics such as the number of sources the query spans, the type of triple pattern joins (start, path, and hybrid [Saleem et al., 2013b]), the use of different SPARQL clauses, the number of query triple patterns, shared variables, etc for SPARQL query generation. Among various metrics in Chapter 4 that

[*] Parts of this chapter have been published as [Rakhmawati et al., 2014c]

can measure performance of the federated engine, we only focus on the data communication cost in this work. The difference between a single RDF repository and a query federation framework is the federation frameworks require communication between the federated engine and a group of SPARQL endpoints. Hence, the cost of data communication is a critical metric to take into account.

8.1 Query Requirements For Federation Framework Benchmark

In this section, we investigate the essential requirements that need to be considered for the generation of queries for assessing federation frameworks. For benchmarking federated SPARQL queries, we draw on existing work from [Montoya et al., 2012c] and [Görlitz et al., 2012]. who provide the following list of query requirements for benchmarking federated SPARQL queries (Number 1 to 4). Requirement number 5 (Object value types) is our contribution.

1. *Various Dataset.* It should be possible to generate queries from any given dataset. If we implement a new system, it is better to test the system with our own dataset in order to close to a real implementation.

2. *Number of source span.* A federated SPARQL query is one that span over at least two data sources. Therefore, the benchmark queries should span over varying number of sources, i.e., between two up to total number of sources in benchmark.

3. *Complexity.* Query complexity can be defined in terms of the total number of triple patterns used, use of different SPARQL clauses, and the type of joins (e.g., subject-subject, subject-object etc.) used between query triple patterns. A benchmark query set should have varying number of triple patterns and should cover maximum SPARQL clauses and the type of joins between triple patterns.

4. *Selectivity/Result-set Size.* The query result-set size depends upon the size of datasets and the selectivity of triple patterns. The benchmark queries should have varying result-set sizes ranging from small to big data queries. A query with low selectivity can consume a lot of bandwidth during query execution since a high number of rows may be retrieved.

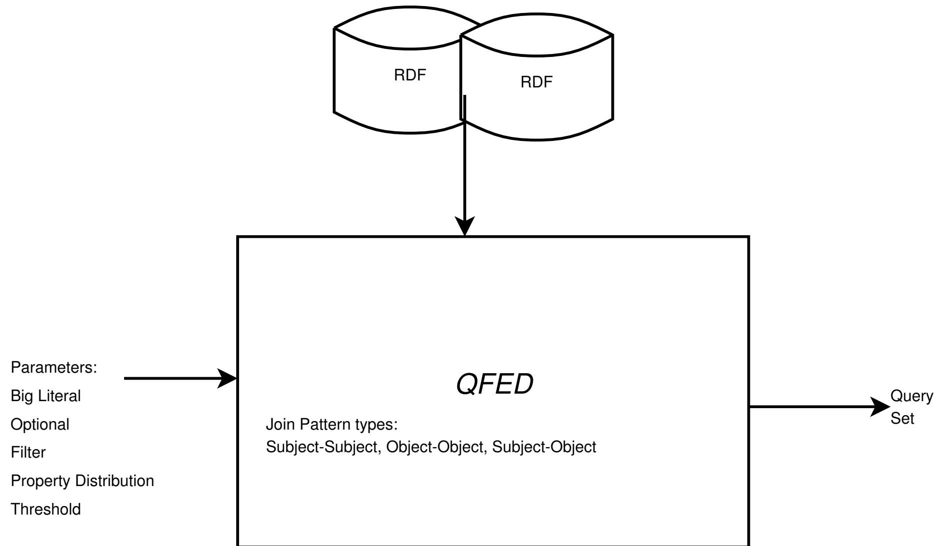


Figure 8.1: QFed Input Output Diagram

5. *Object value types.* With respect to the communication cost, we add object value types as one of the requirements. An object can be either an URI, a blank-node or a literal. A literal object typically has more characters than a URI object. Consequently, querying such literal objects can lead to increased bandwidth usage.

8.2 Query Set Generation

8.2.1 Methodology

To generate queries, we use the following method: Given a set of sources, we create query templates based on particular join patterns. Since we don't only want to create a join between sources, we also expand the query by adding more triple patterns. In order to add those triple patterns and to take into account the cost of data communication, we use two predicate selection strategies: *not considering the property distribution* and *considering the property distribution*. To further increase the communication cost, we also use big literal object values (explained in Equation 8.1). Furthermore, in order to change the selectivity value of a query, we cover two widely used keywords: FILTER and `OPTIONAL`. Additionally, we analyse the effect of these keywords on the data communication cost. In short, the input and output QFed diagram can be found in Figure 8.1.

In summary, our query set generation steps are described as follows:

1. In the data preprocessing stage, we calculate relevant parameters for query set gener-

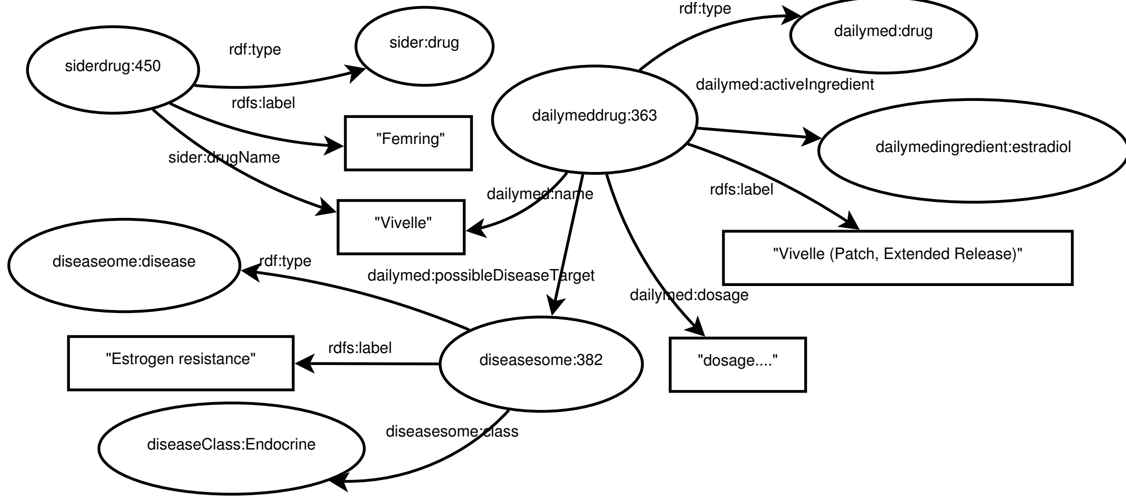


Figure 8.2: Dataset Example

ation, namely the occurrence and frequency.

2. We identify a list of joins that can be found between two entities from two different sources based on the subject-object join pattern, the subject-subject join pattern and the object-object join pattern. After that, we create a set of query patterns between the two entities which form that join pattern. This step will be explained in Section 8.2.3.

3. We add more query patterns to create a star shape where the second entity from the previous step becomes the centre of the shape. The star shape is useful to retrieve a list of information related to the second entity. The query patterns that are added to the query consist of a URI and a literal object value. The predicates of the query patterns are selected based on two methods that will be explained in Section 8.2.4.

8.2.2 Data Preprocessing

Before the query set generation, we pre-process all datasets involved to determine the characteristics of the datasets. We calculate the occurrences of predicates and classes as follows:

Definition 8.1 *Let D be a set of sources that are used in the federation framework, let P be a set of known predicates and let C be a set of known classes. Then the occurrence $\theta P(p, D)$ of predicates p in the dataset D is computed as*

$\theta P(p, D) = |\bigcup_{\forall d \in D} \{d | p \in P(d)\}|$

and the occurrence $\theta C(c, D)$ of class c in the dataset D is calculated as

$\theta C(c, D) = |\bigcup_{\forall d \in D} \{d | c \in C(d)\}|$

Note that we calculate the occurrence of classes and predicates regardless of the frequencies of classes and predicates in each data-source. Our occurrence calculation aims to measure how the predicate and class are spread across the datasets; because a federated engine — the query mediator — generally uses a data catalogue which contains a list of predicates and classes. In other words, in this work, the occurrence is related to the distribution of properties and classes. Based on our previous work in Chapter 7, if the classes and predicates are distributed over the dataset, the federated engine will send a request to more than one dataset in order to detect the most relevant sources for a subquery. In addition, we compute the total frequencies of predicates and classes in the dataset since they are highly associated with the number of intermediate results that are received by the federated engine during query execution.

Definition 8.2 *Given dataset $D = \{d_1, d_2, \cdots, d_n\}$, the frequency $fC(c, D)$ of class c in the dataset D is defined as*

$$fC(c, D) = \sum_{d \in D} \sum_{t \in d} \begin{cases} 1 & \text{if } \exists s : t = (s, rdf{:}type, c) \\ 0 & \text{otherwise} \end{cases}$$

Definition 8.3 *Given dataset $D = \{d_1, d_2, \cdots, d_n\}$, the frequency $fP(p, D)$ of predicate p in the dataset D is defined as*

$$fP(p, D) = \sum_{d \in D} \sum_{t \in d} \begin{cases} 1 & \text{if } \exists s, o : t = (s, p, o) \\ 0 & \text{otherwise} \end{cases}$$

Consider Figure 8.2 as a dataset example, $\theta P(\texttt{rdfs:label}, D)$ and $fP(\texttt{rdfs:label}, D)$ is equal to 3, while $\theta C(\texttt{dailymed:drug}, D)$ and $fC(\texttt{dailymed:drug}, D)$ is equal to 1.

8.2.3 Query Join Pattern Types

A BGP (Section 2) might share one or more of the same variables with other BGPs either in the subject, predicate or object position. These shared variables create a conjunctive query. There are six types of join triple patterns based on the position of the variable in the pattern: Subject-Subject, Predicate-Predicate, Object-Object, Subject-Predicate, Subject-Object, and Predicate-Object. The following are examples of join triple patterns:

- *Subject-Subject* The two following BGPs share variable in the subject position (`?s`).

```
select *
{
    ?s dailymed:possibleDiseaseTarget diseasesome:382 .
```

```
    ?s rdfs:label ?o .
}
```

- *Predicate-Predicate* The following SPARQL query aims to retrieve two different subjects with the same predicate (`?p`).

```
select *
{
    { ?s1 ?p "Vivelle" . }
    UNION
    { ?s2 ?p "Femring" . }
}
```

- *Object-Object* The following SPARQL query aims to retrieve two different subjects with the same object (`?o`).

```
select *
{
    ?s1 sider:DrugName ?o.
    ?s2 rdfs:label ?o .
}
```

- *Subject-Predicate* The following SPARQL query finds a property (`?sp`) that is defined as `rdfs:property`.

```
select *
{
    ?sp a rdfs:property .
    ?s ?sp ?o .
}
```

- *Subject-Object* The following SPARQL query connects two BGPs by using variable in the object position (`?o`).

```
select *
{
    ?s dailymed:possibleDiseaseTarget ?o .
    ?o rdfs:label ?label .
```

```
}
```

- *Predicate-Object* The following SPARQL query connects two BGPs by using variable in the object position and predicate position.

```
select *
{
    ?s1 ?OP ?o .
    ?s2 ?p ?OP .
}
```

In this work, we only consider Object-Object, Subject-Subject, Subject-Object join types since real world queries rarely join a shared variable in the predicate position [Arias et al., 2011]. We will explain how we create these three join patterns as follows:

1. **Subject-Object**

   In federated SPARQL queries, the Subject-Object join pattern is normally used for discovering a relationship between two datasets. This join pattern uses links amongst datasets (Chapter 5) which are generated by publishers. An example is given by `owl:sameAs` and `dailymed:possibleDiseaseTarget` in Figure 8.2. In the context of a federated SPARQL query, a link refers to a predicate which joins two entities that are located in different sources (Definition 5.2). In this chapter, we identify the two entities (s_1 and s_2) that are connected by the link. More precisely:

   Definition 8.4 *Let D be a dataset and $d_1, d_2 \in D$ be two different data sources. Then the set of triples $SO(d_1, d_2)$ is formally defined as follows:*

   $SO(d_1, d_2) = \{(s_1, p_1, s_2) | (s_1, p_1, s_2) \in d_1 \wedge \exists p_2, o_2 (s_2, p_2, o_2) \in d_2\}$

   Informally the subject/object join pattern contains all triples from d_1 that contains a link that joins entity s_1 in source d_1, such that s_1 connects to an entity s_2 in source d_2.

   For subject-object joins, we provide two templates: 1) joining two classes (Figure 8.3(a)), and 2) joining an entity with a class (Figure 8.3(b)). The first query template is a low selectivity query because it maps all entities that are instance of class c_1 in source d_1 to all entities in the source d_2. c_1 is one of the classes in source d_1. Based on those templates, we create a new query in two steps. In the first step, we

identify two entities from two different sources that belong to $SO(d_1, d_2)$. After that, we create query patterns that connect those two entities as shown in Algorithm 8.1. The second step is a process to create a star shape where s_2 is the center of the star shape. The step will be described later in Section 8.2.4. The big literal object will not be added in the first template, thus we disable bigliteral option in the star function.

The following query pattern is an example of the first query template which joins all entities in class `dailymed:drug` with entities in class `diseasome:disease` because the value of the object are guaranteed to be instance of the `diseasome:disease`.

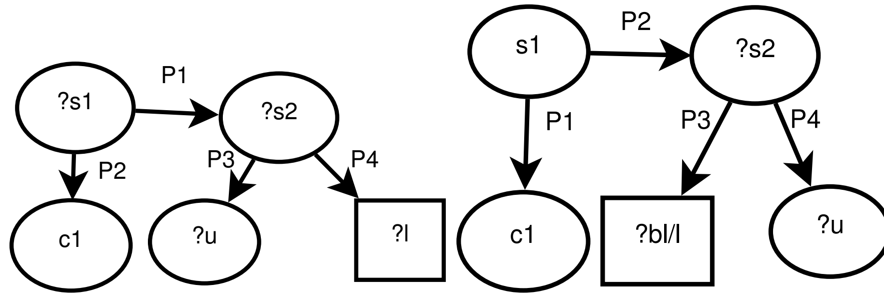
```
?vs1 a dailymed:drugs .
?vs1 dailymed:possibleDiseaseTarget ?vs2 .
```

We only generate the first query template if we discover more than one triples in $SO(d_1, d_2)$ which the subjects are instance of the same class. According to [Arias et al., 2011], the pattern Constant subject-Constant predicate-Variable Object is widely used in DBPedia queries. Therefore, we also create the second query template with the aim of joining an entity in a source to other classes in other sources. The query pattern shown below provides the second template that connects entity `dailymeddrug:363` to all entities in `diseasome:disease`.

```
dailymeddrug:363 dailymed:possibleDiseaseTarget ?s2 .
```

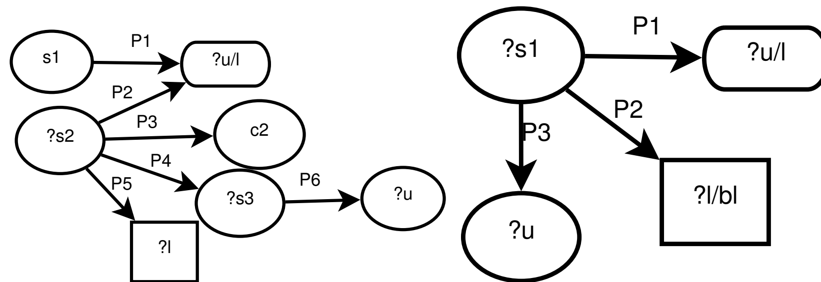
Algorithm 8.1: Subject-object Join Pattern

Require: d_1, d_2
Require: $isbigliteral$
Require: $isdistribution$ /* *the first step* */
1: **for all** $(s_2, rdftype, c_2) \in d_2$ **do**
2:   **for all** $(s_1, rdftype, c_1) \in d_1$ **do**
3:     **if** $(s_1, p_1, s_2) \in SO(d_1, d_2)$ **then**
4:       q1="?vs1 rdf:type "+ c_1". ?vs1 p_1 ?vs2 ." /* *the first template* */
5:       q2="s_1 "+"p_1 ?vs2 ." /* *the second template* */
/* *the second step* */
6:       q1=q1+star(false,$isdistribution$,s_2)
7:       q2=q2+star($isbigliteral$,$isdistribution$,s_2)
8:     **end if**
9:   **end for**
10: **end for**



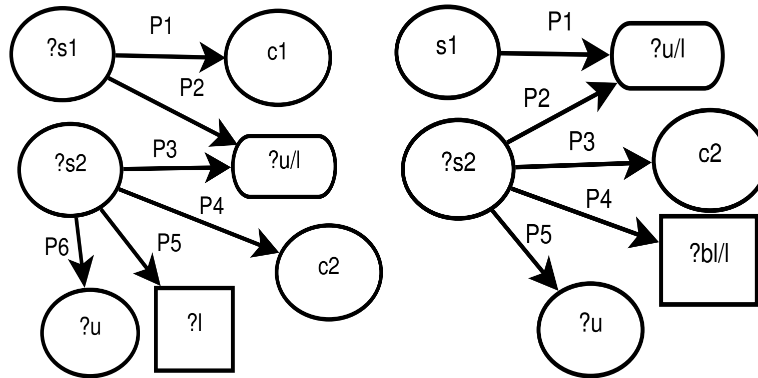
(a) Join two classes in the Subject-Object Join Pattern

(b) Join an entity with a class in the Subject-Object Join Pattern



(c) Hybrid Join Pattern

(d) Join two datasets with a Subject-Subject Join Pattern



(e) Join two classes in the Object-Object Join Pattern

(f) Join an entity with a class in the Object-Object Join Pattern

Figure 8.3: Federated Query Templates

2. **Object-Object**

In order to deal with the problem of lack of availability of interlinks amongst sources, we also generate a query that merges two sources by using an object comparison. We create an object-object join pattern from two sources where two triple patterns share the same variable. We initially construct a list of quadruples $\mathcal{OO}(D)$ each made up of two pairs: a pair consisting of a subject and an object that are located in a single

source and a pair consisting of a subject and an object from another source that have the same object value.

Definition 8.5 *Let D be a finite set of data-sources in a federation framework. Then the list of quadruples $\mathcal{OO}(D)$ for creating an object-object join pattern is formulated as:*

$\mathcal{OO}(D) = \bigcup_{d_1,d_2 \in D}\{(s_1, p_1, s_2, p_2) | \exists o_1(s_1, p_1, o_1) \in d_1 \wedge \exists o_1(s_2, p_2, o_1) \in d_2\}$

Algorithm 8.2: Object-object Join Pattern

Require: d_1, d_2
Require: $isbigliteral$
Require: $isdistribution$
1: **for all** $(s_1, rdftype, c_1) \in d_1$ **do**
2: **for all** $(s_2, rdftype, c_2) \in d_2$ **do**
3: **if** $(s_1, p_1, s_2, p_2) \in OO(D)$ **then**
4: q1="?vs1 rdf:type "+c_1+". ?vs1 "+p_1+" ?o . ?s2 rdf:type "+c_2+". ?vs2 "+p_2"" ?o ." /* *first template* */
5: q2=s_1+" "+ p_1+" ?o . ?vs2 rdf:type "+c_2+". ?vs2 "+p_2+" ?o ." /* *second template* */
6: q1=q1+star(false,$isdistribution$,s_2)
7: q2=q2+star($isbigliteral$,$isdistribution$,s_2)
8: **end if**
9: **end for**
10: **end for**

Like the Subject-Object join pattern, we propose two query templates: 1) join two classes (c_1 and c_2) from different sources (Figure 8.3(e)) and 2) join an entity (s_1) in a source to a class c_2 in another source. As shown in Algorithm 8.2, we also perform two steps. The first step constructs the object-object join pattern, while the second step creates the star shape. The following triple patterns is an example of the first template query for object-object join pattern. All entities that are instances of the class `sider:drug` are mapped to entities that are instances of class `dailymed:drugs`, where `sider:drugName` object's value is the same as dailymed:name object's value.

?vs1 **a** sider:drug .
?vs1 sider:drugName ?o .
?vs2 **a** dailymed:drugs .
?vs2 dailymed:Name ?o .

In the second template, the first line of the triple patterns shown above is removed and variable $?vs1$ is replaced by `siderdrug:450`. Hence, the selective value of the following query that is developed from the second template is higher than the selective value of the first template.

```
siderdrug:450 sider:drugName ?o .
?vs2 a dailymed:drugs .
?vs2 dailymed:Name ?o .
```

3. **Subject-Subject**

   Having the same subject located in multiple sources normally happens when a source is divided into several partitions for a special reason such as data clustering. A subject-subject join pattern is usually used for smushing data[1] that has the same identifier. To begin with, we create a list of subjects $\mathcal{SS}(D)$ that are located in multiple sources as follows:

   Definition 8.6 *Let D be a finite set of data-sources in a federation framework. Then the list of subjects $\mathcal{SS}(D)$ for creating a subject-subject join pattern can be formulated as:*

   $$\mathcal{SS}(D) = \bigcup_{d_1,d_2 \in D}\{s_1 | \exists p_1, o_1(s_1, p_1, o_1) \in d_1 \wedge \exists p_2, o_2(s_2, p_2, o_2) \in d_2\}$$

Algorithm 8.3: Subject-subject Join Pattern

Require: D, d_1, d_2
Require: *isbigliteral*
Require: *isdistribution*
1: **for all** $s_1 \in \mathcal{SS}(D)$ **do**
2:   **for all** $(s_1, p_1, o_1) \in d_1$ **do**
3:     **for all** $(s_1, p_2, o_2) \in d_2$ **do**
4:       q1="?vs1 "+p_1+" ?o1. ?vs1 p2 ?o2 ."
5:       q1=q1+star(*isbigliteral*,*isdistribution*,s_2)
6:     **end for**
7:   **end for**
8: **end for**

4. **Hybrid Join**

[1] RDFSmushing: `http://www.w3.org/wiki/RdfSmushing`

The aforementioned query templates only merge data from two sources. To merge data from more than two sources, we combine the object-object join pattern query template and subject-object query template. In the first step, we create a similar query template to the path that joins an entity to a class in the object-object template (Figure 8.3(b)). Then, we iterate each predicate of s_2 in the $PObjType(s_2, d_2)$ to find the predicate that also belongs to $\mathcal{SO}(d_2, d_3)$ (Algorithm 8.4 line 4-9).

Algorithm 8.4: Hybrid Join Pattern

Require: d_1, d_2, D
Require: $isbigliteral$
Require: $isdistribution$
1: **for all** $(s_1, rdftype, c_1) \in d_1$ **do**
2: **for all** $(s_2, rdftype, c_2) \in d_2$ **do**
3: **if** $(s_1, p_1, s_2, p_2) \in OO(D)$ **then**
4: **for all** $p_3 \in PObjType(s_2, d_2)$ **do**
5: **for all** $d_3 \in D$ **do**
6: **if** $d_3 \neq d_2 \wedge d_3 \neq d_1$ **then**
7: $s_3 = \{s_3 | (s_2, p_3, s_3) \in \mathcal{SO}(d_2, d_3) \wedge s_3 \neq s_1\}$
8: **if** $s_3 \neq \emptyset$ **then**
9: q1=s_1+" "+ p_1+" ?o . ?vs2 rdf:type "+c_2+". ?vs2 "+p_2+" ?o . ?vs2 "+ p_3+" ?vs3"
10: q1=q1+star(false,$isdistribution$,s_2)
11: q1=q1+starH($isdistribution$,s_3)
12: **end if**
13: **end if**
14: **end for**
15: **end for**
16: **end if**
17: **end for**
18: **end for**

In order to avoid a reciprocal query to the first source, we reject a predicate whose object equals to entity s_1 which is the subject of the first source. Let s_3 be an object of a triple in source d_2 whose predicate is an element of $\mathcal{SO}(d_2, d_3)$, then we use the same procedure (Section 8.2.4) to select a predicate that will be the part of the final query triple pattern. For the final query pattern, we only choose a predicate with the literal object value or URI object value to reduce the complexity of the query.

8.2.4 Star shape Creation

After creating the join patterns above, we add triple patterns for creating a star shape by using the star function (Algorithm 8.5). In order to create the star shape, we perform two steps: 1) identify object type of the predicate and 2) select the suitable predicate that will be added.

Object Value Types

There are three types of objects that are defined in our works: u for URI, l for literal and bl for big literal. A big literal object (bl) is a literal which has more characters than a predefined value Z. The value of Z can vary depending on the dataset. The motivation for adding a big literal object value in the query pattern is to assess how to optimise a federated engine to deal with the cost of data communication. The function $typef(o, D)$ is a function to decide the type of a value of object o in dataset D which is defined as follows:

$$typef(o, D) = \begin{cases} u & \text{if } o \in U \\ l & \text{if } o \in L \wedge length(o) < \text{Z} \\ bl & \text{otherwise} \end{cases} \tag{8.1}$$

Let s be a subject in source d, the set of pairs of predicates and its object value type $PObjType(s, d)$ can be formulated in Equation 8.2.

$$PObjType(s, d) = \{(p, typef(o, d)) | \exists (s, p, o) \in d \wedge p \neq \text{rdf:type}\} \tag{8.2}$$

For instance, as shown in Figure 8.2, `diseasesome:382` has two properties: `rdfs:label` and `diseasesome:class` (except for `rdf:type`). Thus, $PObjType($`diseasesome:382`$, d)$ is {(`rdfs:label`,l),(`diseasesome:class`,u) }.

The Predicates Selection

The objective of the predicate selection is to find suitable predicates for triple query patterns that are added to the query. We propose two approaches for predicate selection: 1) *not considering the property distribution* (ND) 2) *considering the property distribution* (D). In the first approach, we just do an iteration for each pair of the predicate and its object type for entity s in $PObjType(s, d)$ (see Algorithm 8.5 line 4-17). The iteration will stop once we get a predicate with a URI object value and a pred-

Algorithm 8.5: Star Function

```
Require: isbigliteral                                   /* bigliteral flag */
Require: isdistribution                     /* property distribution flag */
Require: d                                                     /* source */
Require: s                                                     /* entity */
 1: q=””
 2: blvalue=lvalue=uvalue=false
 3: if (isdistribution=false) then
 4:    for all (p, type) ∈ PObjType(s, d) do
 5:       if (type=b and isbigliteral=true and blvalue=false) then
 6:          q=q+”?s2 p ?bl”
 7:          blvalue=true
 8:       else if (type=l and lvalue =false) then
 9:          q=q+”?s2 p ?l”
10:          lvalue=true
11:       else if (uvalue=false) then
12:          q=q+”?s2 p ?u”
13:          uvalue=true
14:       else if ((blvalue=true or lvalue=true) and uvalue=true ) then
15:          break looping
16:       end if
17:    end for
18: else
19:    pu = 0                  /* the distribution of property for a URI object */
20:    pl = 0              /* the distribution of property for a literal object */
21:    pb =0           /* the distribution of property for a big literal object */
22:    blvalue=lvalue=uvalue=””
23:    for all (p, type) ∈ PObjType(s, d) do
24:       pd=distribution(p)
25:       if (type=bl and isbigliteral=true and pd > pb) then
26:          blvalue=p
27:          pb=pd
28:       else if (type=l and pd > pl) then
29:          lvalue=p
30:          pl=pd
31:       else if (type=u and pd > pu) then
32:          uvalue=p
33:          pu=pd
34:       end if
35:    end for
36:    if (isbigliteral=false) then
37:       q=”?s2 pl ?l . ?s2 pu ?u .”
38:    else
39:       q=”?s2 pb ?bl . ?s2 pu ?u . ”
40:    end if
41: end if
42: return q
```

Algorithm 8.6: Star Function for Hybrid Join

```
Require: isdistribution                          /* property distribution flag */
Require: d                                                         /* source */
Require: s                                                         /* entity */
 1: q=""
 2: lvalue=false
 3: uvalue=false
 4: if isdistribution=false then
 5:    for all (p, type) ∈ PObjType(s, d) do
 6:       if (type=l and lvalue =false) then
 7:          q=q+"?s2 p ?l"
 8:          lvalue=true
 9:       else if (type=u and uvalue=false) then
10:          q=q+"?s2 p ?u"
11:          uvalue=true
12:       else if (lvalue=true) or uvalue=true) then
13:          break looping
14:       end if
15:    end for
16: else
17:    pu = 0                 /* the distribution of property for a URI object */
18:    pl = 0             /* the distribution of property for a literal object */
19:    lvalue=""
20:    uvalue=""
21:    for all (p, type) ∈ PObjType(s, d) do
22:       pd=distribution(p)
23:       if (type=l and pd>pl) then
24:          lvalue=p
25:          pl=pd
26:       else if (type=u and pd>pu) then
27:          uvalue=p
28:          pu=pd
29:       end if
30:    end for
31:    if pl¿pu then
32:       q="?s2 pl ?l ."
33:    else
34:       q="?s2 pu ?u . "
35:    end if
36: end if
37: return q
```

icate with a literal object value. If the big literal option is enabled, we iterate over $PObjType(s, d)$ until we discover a predicate with the big literal object value and a predicate with the URI object value. If we cannot find a predicate with the big literal object value, we choose a predicate with literal object value instead. For instance, we create a subject-object query from the Dailymed and Diseasome sources based on the data from Figure 8.2. The results of $SO(Dailymed, Diseasome)$ is { `dailymeddrug:363`, `dailymed:possibleDiseasetarget`, `diseasome:382` }. Then, we find all predicates that belong to `diseasome:382` by using $PObjType($`diseasome:382` $, Diseasome)$. Suppose we obtain (`diseasome:size`,l), (`rdfs:label`,l), (`diseasome:class,u`) sequentially, we only choose `diseasome:size` and `diseasome:class`. The `rdfs:label` is not selected since its position is after the `diseasome:size` position. As a result, we generate the query shown in Listing 8.1.

Listing 8.1: Example of a Subject-Object join pattern Query

```
select * {
       ?vs1 a dailymed:drugs .
       ?vs1 dailymed:possibleDiseaseTarget ?vs2 .
       ?vs2 diseasome:class ?u .
       ?vs2 diseasome:size ?l .
 }
```

Listing 8.2: Example of an Object-Object join pattern query

```
select * {
       ?vs1 a sider:drug .
       ?vs1 sider:drugName ?o .
       ?vs2 a dailymed:drugs .
       ?vs2 dailymed:Name ?o .
       ?vs2 dailymed:activeIngredient ?u .
       ?vs2 dailymed:dosage ?bl .
 }
```

The second approach chooses the predicate p with the highest occurrence $\theta P(p, D)$ since federated engines generally exploit a data catalogue that contains the list of predicates to select a relevant source for a query. Using predicates that are distributed over the dataset will increase the cost of data communication between the federated engine and SPARQL endpoints [Rakhmawati et al., 2014b]. Suppose we create a feder-

ated query by implementing an object-object join pattern from the Sider and Dailymed sources in Figure 8.2. The $\mathcal{OO}(D)$ produces {`siderdrug:450`, `sider:drugName`, `dailymeddrug:363` ,`dailymed:Name` } since `sider:drugName` and `dailymed:Name` have the same objects value (*Vivelle*). As the next step, we find all predicates for entity `dailymeddrug:363` and their object value type: $PObjType(dailymeddrug : 363, Dailymed)$. We get $PObjType$(`dailymeddrug:363`, $Dailymed$) = {(`dailymed:activeIngredient`, u), (`rdfs:label`, l), (`dailymed:dosage`, bl), (`dailymed:possibleDiseaseTarget`, u)}. In this query generation, we pick `dailymed:activeIngredient` and `dailymed:dosage`, since we consider the predicate occurrence and the big literal option. Although `rdfs:label` occurrence is higher than `dailymed:dosage`, the `dailymed:dosage` is preferred to `rdfs:label` since we give higher priority to the predicate with the big literal than to the predicate with the higher predicate occurrence.

In the next example, we extend our previous federated query example to create a hybrid join query. One element of $PObjType$(`dailymeddrug:363`, $Dailymed$) is `dailymed:possibleDiseaseTarg` which is also a predicate that points to `diseasome:382` at the Diseasome source. Therefore, we can create a subject-object join pattern by using `dailymed:possibleDiseaseTarget`. As shown in the first example, we retrieve values of $PObjType$(`diseasome:382`, $Diseasome$) = {(`diseasome:size`, l), (`rdfs:label`, l), (`diseasome:class`, u)}. Since θP(`rdfs:label`) is greater than θP(`diseasome:size`) and θP(`diseasome:class`), we choose `rdfs:label` to be the predicate for the triple pattern with a URI object value. Note that, for the hybrid join, we disregard the type of object value (see Algorithm 8.6). Furthermore, in order to simplify the query, we do not select a predicate with the big literal value.

The last example is a subject-subject join pattern federated query. Suppose that the related triples with `dailymeddrug:363` as the subject that are located in two different sources, `(dailymeddrug:363,dailymed:name,"Vivelle")` is in the first source and the rest of the triples are in the second source. If we also enable big literal parameters, then we can create the query shown in Listing 8.3.

Listing 8.3: Example of a Subject-Subject join pattern query

```
select * {
       ?s dailymed:Name ?l .
       ?s dailymed:possibleDiseaseTarget ?u .
       ?s dailymed:dosage ?bl .
 }
```

Listing 8.4: Example of a hybrid join pattern query

```
select * {
       disease:382 sider:drugName ?o .
       ?s2 a dailymed:drugs .
       ?s2 dailymed:Name ?o .
       ?s2 rdfs:label ?LITERAL .
       ?s2 dailymed:possibleDiseaseTarget ?s3 .
       ?s3 rdfs:label ?LITERAL3 .
}
```

Listing 8.5: Example of a federated SPARQL query using the FILTER and `OPTIONAL` Keywords

```
select * {
       ?s1 a dailymed:drugs .
       ?s1 dailymed:possibleDiseaseTarget ?s2 .
       ?s2 diseasome:class ?u .
       OPTIONAL{ ?s2 diseasome:size ?l . }
       FILTER (?l >= 1)
}
```

8.2.5 Queryset Generation Extension

Queryset Threshold

Since the number of entities is quite large, we limit the number of queries based on two parameters: a) the frequency of predicates ($Fp(p, D)$) and b) the number of entities in each class. The goal of the first parameter is to take a subset of $SO(d_1, d_2)$, $\mathcal{OO}(D)$ or $\mathcal{SS}(D)$

Listing 8.6: Example of a federated SPARQL Query using the SERVICE keyword

```
select * {
SERVICE<http://localhost/dailymed/sparql> {
       ?s1 a dailymed:drugs .
       ?s1 dailymed:possibleDiseaseTarget ?s2 . }
SERVICE<http://localhost/diseasome/sparql> {
       ?s2 diseasome:class ?u .
       ?s2 diseasome:size ?l .
} }
```

that will be processed in the query generation. The subset consists of the top-K predicates with the highest $Fp(p, D)$. We expand the results of the first parameter into several query generations. However, we give the second parameter as a constraint to limit the number of entities of each class. We restrict only n entities for each class with the same predicates to be generated since several entities in the same class may have the same predicate, which is also part of $SO(d_1, d_2)$ or $\mathcal{OO}(D)$.

A predicate that belongs to an entity does not always belong to another entity, even when they are in the same classes. In order to retrieve more query results, we provide the `OPTIONAL` keyword that is inserted in the one of triple patterns that is associated to entity s'. The `OPTIONAL` and `FILTER` keywords are only applicable for joining inter-class query templates (Figure 8.3(a) and 8.3(e)) since those two templates cover all entities in the same class. We only apply the `FILTER` keywords in queries that contain a literal with integer value. To find the constraint value for a `FILTER` expression, we choose the median value of a set of literal answers. Then, we use the greater than or equal to sign ($\geq$) in the `FILTER` expression. An example of a query using the `OPTIONAL` and `FILTER` keywords can be found in Figure 8.5.

The SPARQL 1.1 standard is supported in most SPARQL endpoint servers (Chapter 3). We can execute federated SPARQL queries by using the *SERVICE* keywords in SPARQL 1.1 standard. Therefore, we also provide a query that contains the SERVICE keyword to asses the federated engine that supports SPARQL 1.1 features. This implies that our query can be executed in a federated engine that does not support a transparent query interface.

8.2.6 Comparison of Splodge, Lidaq and QFed

To the best of our knowledge FedBench [Schmidt et al., 2011b] is the only SPARQL query federation benchmark; it provides 16 static queries for querying multiple static real sources (Chapter 4). DAW [Saleem et al., 2013b] provides a set of static queries based on characteristics of BSBM queries [Bizer and Schultz, 2009] from four public datasets. The queries cover most SPARQL operators and keywords. However, all the queries are statically generated and are not complex enough to match the real world queries (maximum of 4 triple patterns per query). SPLODGE [Görlitz et al., 2012] offers a tool that can generate a query set based on some characteristics of the datasets. LidaQ [Umbrich et al., 2012a] also provides a query set generation tool for federated query based on three query templates. Both SPLODGE and LidaQ rely on the links (e.g `owl:sameAs`) between datasets. Since not all entities are interlinked with each other, we propose a query set generator that does not only use links

	Lidaq	Splodge	QFed
Join pattern between different datasets	S-O	S-O	S-S,O-O,S-O
Maximum Triple patterns	3	unlimited	7
Predicate	Unbound and bound predicates	Bound predicate	Bound predicate
Keywords Supported	✗	✗	Filter, Optional, Service
Object value types	✗	✗	Literal, URI
Selectivity	✗	✓	✓
Number of sources	✗	✓	✓

Table 8.1: Features and Capabilities of QFed, Lidaq and Splodge. S=Subject,O=Object

like Lidaq and SPLODGE, but also uses object and subject comparisons between two and three sources. SPLODGE generates unlimited triple patterns since it attempts to link all sources in a dataset. However, it does not reflect a real world query. Therefore, QFed is designed to create a query with a limited number of queries (maximum 7 triple patterns). Table 8.1 compares the features and capabilities of QFed, Lidaq and Splodge. As shown in the table, these query generators produce different queries. Hence, in our evaluation, we will not compare the performance of these query generators in producing queries.

8.3 Evaluation

The first objective of our evaluation is to show that our generated queries can return results that involve more than one data source. Another objective is to demonstrate the effect of: 1) the big literal object value, 2) predicates occurrence, 3) `FILTER` keyword, and 4)`OPTIONAL` keyword on the performance of a federated engine especially on cost of data communication. The results of this evaluation can give a consideration how to create a query for assessing a performance of a federated engine.

8.3.1 Experimental Setup

This section describes the evaluation system for running our query set on the federated engines. The code for generating the queries can be found at `https://github.com/nurainir/QFed`.

Dataset

Our dataset consists of four life science datasets: Dailymed, Drugbank, Diseasesome and Sider. These datasets are interlinked with each other using a number of predicates, as previously shown in Figure 3.2. Additionally the same object values can be found in these datasets; these values, such as `rdfs:label`, `dailymed:name` and `drugbank:interactionDrug1`, can be used federated SPARQL queries. Due to the problem of unavailability of triples with the same subject at different datasets, we divide the Drugbank dataset into two partitions. We distributed the triples that are related to class *Drug* to all partitions evenly. We then add the triples that contain *drug_interactions*, *references* and *Enzim* classes to partition one, and add the rest of the triples to partition two. In total, we have 5 SPARQL endpoints for accessing five data-sources.

System

We set up five Fuseki[2] engines on a Linux virtual machine for storing five datasets. We bound Fuseki to five different ports. The evaluation result will not be influenced by the machine because we only measure the data throughput of the federated engine and SPARQL endpoint. On a separate virtual machine, we installed FedX[Schwarte et al., 2011] and Fuseki as the federated engines. In this evaluation, we do not compare the performance of Fuseki and FedX. We run queries containing the `SERVICE` keyword on Fuseki, whereas queries without the `SERVICE` keyword are executed on FedX. We choose FedX as the federated engine since it is able to support most of the operators and keywords in the SPARQL 1.1 standard.

Metrics

We assess the performance of FedX and Fuseki based on the following two metrics: data transmission, and run time. The data transmission refers to the amount of data sent and received between the federated engine and SPARQL endpoints measured in byte during query execution. The run time is started when the federated engine receives a query from the client and ended when it dispatches the query results to the client. The data transmission is closely related to our goal of investigating the impact of: 1) big literal object values, 2) the distribution of the predicates, and 3) `OPTIONAL` and `FILTER` keywords on the performance of a federated engine.

[2] `http://jena.apache.org/documentation/serving_data/`

Query set	#Queries	#S-S	#O-O	#S-O	#H
Threshold of Classes 2 and Properties 2	159	3	52	98	6
Threshold of Classes 3 and Properties 3	202	3	69	124	6

Table 8.2: The list of Query-set Generation (S-S=Subject-Subject Join,O-O=Object-Object Join,S-O=Subject-Object Join,H=Hybrid Join)

Query set	3 BGPs	4 BGPs	5 BGPs	6 BGPs	7 BGPs
Threshold of Classes 2 and Properties 2	70	35	32	16	6
Threshold of Classes 3 and Properties 3	99	33	48	17	5

Table 8.3: The Number of BGPs for each queries in the query set

Query set

In total, we produced 11552 queries in 64 categories. We distinguish the categories based on the query threshold, the existence of big literal objects value, `OPTIONAL` and `SERVICE` keywords. Although we insert `SERVICE` and `OPTIONAL` keywords, the average predicate occurrence and number of queries stay the same. We only list 2 query set categories in Table 8.2 for simplicity. Table 8.4 provides the statistics of the `OPTIONAL`, `FILTER` and big literal object value usage in the query set. Only a few of the queries contain the `FILTER` keyword, since it only applies to the query pattern containing a literal with an integer value. As shown in Table 8.3, QFed produces queries that consist of three to seven triple patterns. Some examples of the queries can be found in Appendix B.3. Due to the limitations of Fuseki for running a query, we also append LIMIT keywords for the queries that run on Fuseki. Each query is executed three times and is limited to 10 minutes of execution each time.

8.3.2 Results and Discussion

The main goal of including the `SERVICE` keyword in a query is to ensure that the query covers more than one dataset. If Fuseki returns an empty result for a query with the `SERVICE` keyword, this means that the query does not involve more than one dataset. The results of `SERVICE` query execution show that only 0.07% out of the queries generated failed to return answers.

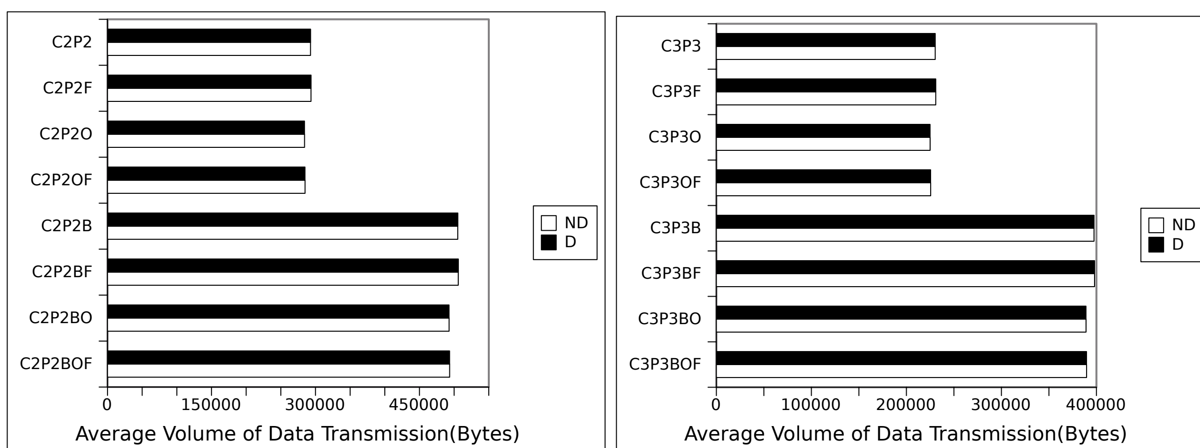
In order to distinguish each query set category, we name the query sets as follows: the Label C following a digit represents the constraint of number entities for each class. By adding the label P and following with the digit, we limit the number of properties based on their frequencies. B, the final label, denotes a query set that contains big literal object

Query Set	OPTIONAL/FILTER/BIGLITERAL
C2P2	0/0/0
C2P2B	0/0/61
C2P2BF	0/5/61
C2P2BO	51/0/61
C2P2BOF	51/5/61
C2P2F	0/5/0
C2P2O	53/0/0
C2P2OF	53/5/0
DC2P2	0/0/0
DC2P2B	0/0/61
DC2P2BF	0/5/61
DC2P2BO	51/0/61
DC2P2BOF	51/5/61
DC2P2F	0/5/0
DC2P2O	53/0/0
DC2P2OF	53/5/0
C3P3	0/0/0
C3P3B	0/0/83
C3P3BF	0/6/83
C3P3BO	50/0/83
C3P3BOF	50/6/83
C3P3F	0/6/0
C3P3O	52/0/0
C3P3OF	52/6/0
DC3P3	0/0/0
DC3P3B	0/0/83
DC3P3BF	0/6/83
DC3P3BO	50/0/83
DC3P3BOF	50/6/83
DC3P3F	0/6/0
DC3P3O	52/0/0

Table 8.4: Statistical queries containing `FILTER`, `OPTIONAL` and big literal object value

values. In our evaluation, we add labels O, F and B to denote query with `OPTIONAL`, `FILTER` keywords and big literal object value respectively. For instance, C2P2B means that the number of entities for each class is two, the number of properties is two and we consider a big literal object value while creating query patterns.

As shown in Figure 8.4, the replacement of the literal object value with the big literal object value led to an increase in the data transmission volume between the Fuseki and SPARQL endpoints. The `OPTIONAL` and `FILTER` keywords do not contribute to increasing the communication cost since we added the `LIMIT` keyword for the query with the `SERVICE` keyword. The `LIMIT` keyword aligns the selectivity values of queries with the same ID among categories. There is not much difference between a query concerning the property distribution(D) and a query that does not take account of the property distribution(ND). The reason is that Fuseki does not have a source selection mechanism; rather the user has to define the source of each subquery beforehand. As such, the distribution of properties does not influence the communication cost.

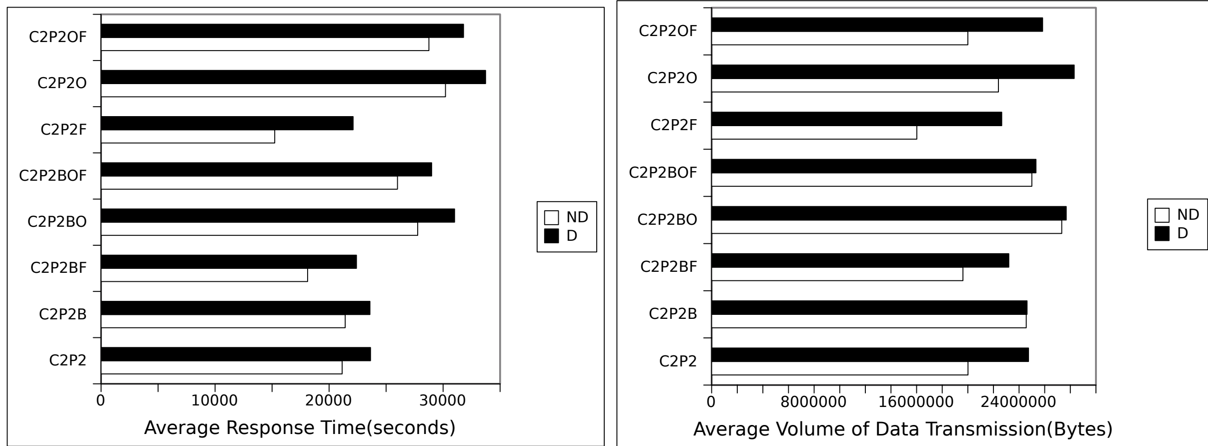


(a) Threshold of Classes 2 and Properties 2 (b) Threshold of Classes 3 and Properties 3

Figure 8.4: Average Volume Data Transmission on Fuseki Execution (in bytes)

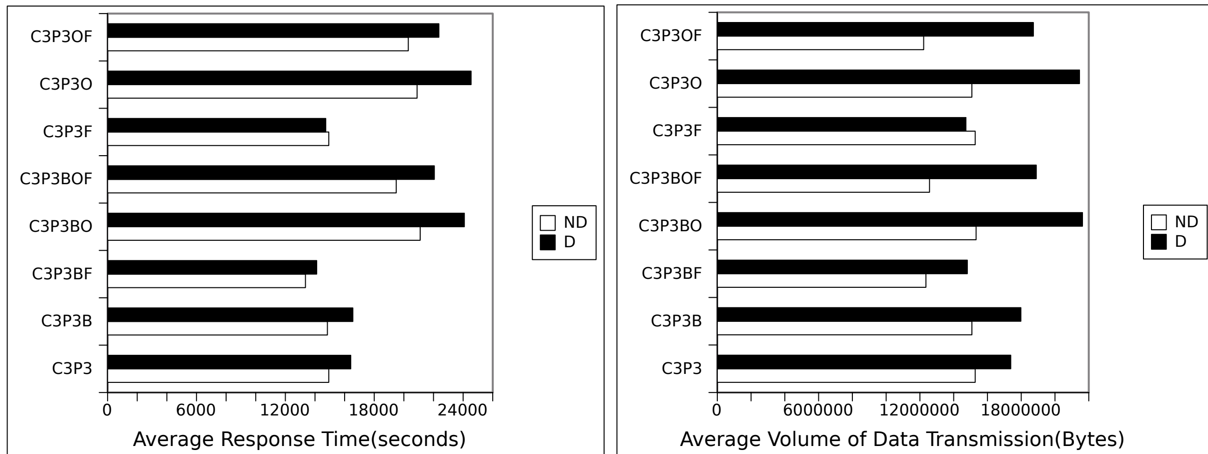
Due to time out issues and Java memory heap space problems, FedX failed to execute 5.68% of the queries. We eliminated all the failed query results along with all the queries that have the same IDs with the failed ones. A selectivity value of the query is low when an `OPTIONAL` keyword is added to the query, as a result, the response time and the data transmission volume increase as shown in Figures 8.5(a)- 8.6(b). The use of big literal objects also causes an increase in data transmission volume (Figures 8.5(b) and 8.6(b)), but if a query with a small literal object has a smaller selectivity value than a query with the big

literal value, then in some cases, they require almost the same bandwidth consumption. If the property distribution is considered, the FedX performance gets worse because FedX has to interrogate more SPARQL endpoints to execute a single query. The addition of a FILTER keyword in a query can cut the number of intermediate results and eventually reduces the communication cost between the FedX and SPARQL endpoints.



(a) Average Response Time (b) Average Data Transmissions

Figure 8.5: FedX Execution Results For Threshold 2



(a) Average Response Time (b) Average Data Transmissions

Figure 8.6: FedX Execution Results For Threshold 3

8.4 Conclusion

This chapter presented QFed, a tool for generating the queries to assess the performance of a federated engine. The query generation strategy considers the distribution of the predicates as a characteristic of the dataset. To measure the cost of data communication, we added big literal objects, and `FILTER` and `OPTIONAL` keywords in queries. In order to integrate data from different sources, we identified subject-object, subject-subject and object-object join patterns. The experimental results show that big literal objects have a significant impact on Fuseki performance since Fuseki executes a query that declares SPARQL endpoints before execution. The use of predicates that are distributed over the dataset, the big literal object, FILTER keyword and OPTIONAL keyword in the query influence the volume of data transmission between the FedX and the SPARQL endpoints and the FedX response time. By adding an OPTIONAL keyword in a query, the performance of FedX is more significantly getting worse than the addition of a big literal value in a query since it can reduce the selectivity value of a query.

Chapter 9

Discussion and Conclusions

The increasing attention on federated SPARQL query systems create the need for benchmarking these systems to evaluate their performance. Compared to traditional relational database benchmarking, federated SPARQL query system benchmarking are still in its infancy. Evaluating a federated SPARQL system is a big challenge since it consists of heterogeneous and distributed systems. Apart from that, performance of the federated SPARQL system can be influenced by of the components in that system. These conditions have motivated this work.

The main research question of this thesis, presented in Section 1.1, asks:

> **How can we design a comprehensive SPARQL query federation benchmark based on the dependencies between the metrics, dataset and queries?**

Summary of Contributions

First of all, in order to benchmark a federated SPARQL system, we need to understand various types of federated systems and the differences between them. Hence, we conducted a survey of the existing federated engines and compared their behaviours and approaches to handling SPARQL queries in Chapter 3. We classified these engines into three architecture categories namely executor, rewriter, and planner architectures.

To address our research question, we described and worked on the three core components of a benchmark system as shown in Figure 1.2, namely metrics, queries and datasets. We describe each of these components in turn.

Metrics

In Chapter 4, we studied the state of the art in both single RDF repository and distributed RDF repositories benchmarks. Our study focused primarily on the metrics, dataset and queries that are used in those benchmark suites. The findings about dataset and queries were used in the subsequent chapters of this thesis. We then discussed the basic performance metrics that can be measured in a federated SPARQL query environment based on our investigation in Chapter 3.

Continuing, based on the data transactions between a federated engine and SPARQL endpoints, we introduced a set of independent metrics that are not influenced by the federated SPARQL query environment. We also investigated semi-independent metrics that take into account SPARQL endpoints as one of the factors that contributes to the performance of a federated SPARQL query framework. These metrics are derived from the independent metrics.

To evaluate the performance of a federated engine, it is necessary to run more than one query. The queries should have various complexity values (see Page 63). In order to avoid trade-off between query set results after merging the results, we proposed a composite metric that assigns different weight values for each query before summing those queries results up into a single value. We then combined different performance metrics into a single performance value by using the geometric mean. The motivation behind this approach is to give a meaningful performance summary and this ultimately helps us to compare amongst the performance of different federated systems.

Dataset

The dataset component is presented in three chapters: Chapters 5, 6 and 7. First, in Chapter 5, we discussed the impact of interlinking between sources on the performance of a federated engine. As one of the characteristics of a dataset, a link can aid in merging data from multiple sources. We investigated the costs and benefits of the presence of the links in a federated SPARQL query by running three differences experiments.

Thereafter, in Chapter 6, we introduced six approaches to partitioning a dataset for benchmarking a federated SPARQL query system; these were inspired from a single RDF repository clustering partition. Those approaches are graph partition, class partition, entity partition, property partition, triples partition and hybrid partition. We also provided a lightweight tool to generate a dataset by using those approaches.

Finally, Chapter 7 presented a notion — called the spreading factor of a dataset — to analyse the distribution of classes and properties throughout the dataset. There are two types of the spreading factor metric: 1) the spreading factor of the whole dataset regardless of the existence of the query set; 2) the spreading factor of a dataset associated with the particular query set that is used in the evaluation. We ran an evaluation to investigate the relationship between the spreading factor and the performance of a federated engine. This evaluation relied on our work in Chapter 6 because we used nine datasets based on our approaches from Chapter 6. Our experimental results showed that the class and property distribution affect the communication cost between a federated engine and the SPARQL endpoints.

Queries

Our final contribution is to generate queries for benchmarking a federated SPARQL query system; this was presented in Chapter 8. We identified the possible patterns used to join data from multiple sources, in particular between two or three sources. The queries are generated based on the characteristics of a dataset as well as the performance metric. With respect to the characteristics of a dataset, we consider the distribution of the properties , and frequencies of the properties as the parameters for query generation. We also cover two widely used keywords in SPARQL queries: `FILTER` and `OPTIONAL`. Those keywords may influence the selectivity of a query. Hence, this eventually affects the volume of data transmissions which is one of the performance metrics for benchmarking a federated SPARQL system. In addition, we include a big literal option for assessing the performance of a federated engine in processing large data. Our experimental results showed that QFed can successfully generate a large set of meaningful federated SPARQL queries that can be used in the performance evaluation of different federated SPARQL query engines.

9.1 Future Directions

In this thesis, we have presented a comprehensive evaluation of the federated SPARQL query system by observing three different benchmark components. Next, we look at what we think are important future directions arising from the work presented in this thesis.

Data Quality

One of parameter of a good benchmark for a database is the quality of the data produced by the system. The speed of a federated engine to return back the query result will not matter, if the query result is not correct. Therefore, we used the number of results and the accuracy as the data quality metrics in Chapter 5. Another data quality metric that should be considered is the freshness of the query results. The freshness-rate is the ratio of extracted elements that are up-to-date, i.e. their result values equal to the SPARQL endpoint [Peralta, 2006]. Although a federated engine accesses data from the SPARQL endpoint directly, the freshness of the query result may be compromised if the data catalogue is not up-to-date. Updating a data catalogue uses an expensive query. Therefore, it generally only updates periodically.

Cover more basic performance metrics

Even though we have mentioned many basic performance metrics in Chapter 4, we did not use all of them such as CPU usage and memory usage in our evaluation since we were concerned with the independent metrics. We only used the response time since it is widely used in most of current evaluations that are reported in Table 4.1.

Another important metric for a querying system is query throughput, which is defined as the number of queries that can be executed in a given period of time. This metric becomes more important once parallel queries are allowed to come through the system at once. The best throughput can be achieved when the system can execute many concurrent tasks in the same time. For the moment, our evaluation does not calculate the query throughput. Rather, we run a single query individually to investigate the effect of each query regardless of other queries. It is difficult to determine the factors that influence the performance while running several queries at once.

SPARQL Endpoint Performance Assessment

Building a holistic benchmark is a hard task. As of this writing, we assess the SPARQL endpoint performance from the federated engine side. Assessing SPARQL endpoint performance is a non-trivial task since it is normally located in different machines. There should be a synchronisation between a federated engine benchmark system and a SPARQL endpoint benchmark system. Further, it is difficult to assess federated SPARQL query performance that uses public SPARQL endpoints as the data provider because we cannot deploy the

SPARQL endpoint benchmark system in a public SPARQL endpoint.

Heterogeneous Systems

Our evaluation used the same software for all SPARQL endpoint servers. Thus, the SPARQL endpoint performance is only influenced by the hardware and characteristics of the dataset that are stored in the server. It will be a large evaluation task to carry out an evaluation with various systems. To investigate the impact of the SPARQL endpoint server, we should conduct a set of experimental systems that represent all combinations of SPARQL endpoint server, hardware and characteristics of the dataset. We can adopt [Kjernsmo and Tyssedal, 2013] to create a set of experiments with various combinations.

Metric

We have introduced the notion of composite metric in Chapter 4 which refers to a metric that merges the performance values for a single metric into a single value. Sometimes, it is hard to compare the performance of different federated engines against each other based on different metrics. Thus, it is necessary to formulate a composite metric that combines different metrics into a single value. It is better to allow the user to assign a weight for each metric before merging them in a single value since choosing a metric is an individual decision.

Query

How to generate queries for benchmarking a federated SPARQL query framework is still an open question. There are some requirements that have not been addressed yet by our system (QFed) such as the number of query patterns. Apart from that, we have primarily focused on the `SELECT` query form which is a read-only operation. Since SPARQL 1.1 also supports update operations such as `INSERT` and `DELETE`, we see a great need for evaluating those update operations queries.

Benchmark Community Acceptance

As we pointed out in Chapter 1, a benchmark should be accepted by the community targeted. There should be a public system that stores the configuration of evaluations. These configurations can be useful for other communities to re-run the same evaluation. As a

result, the community can clarify the results from previous evaluation.

Comparison with other benchmarks

As of this writing, to the best of our knowledge, FedBench is the only completed benchmark system that is intended for a federated SPARQL system. It is worth comparing FedBench to our system while running an evaluation.

Components integration

As reported in Chapter 4, a benchmark suite consists of metrics, queries and dataset. Apart from that, it provides a test driver to help users to assess the performance of a system automatically. At present, we have not integrated the metrics, query generator and dataset generator into a single system yet. Our systems are developed separately.

"If we wait for the moment when everything, absolutely everything is ready, we shall never begin."

—Ivan Turgenev

Bibliography

Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2007). Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 411–422. VLDB Endowment.

Acosta, M. and Vidal, M.-E. (2011). Evaluating adaptive query processing techniques for federations of sparql endpoints. 10th International Semantic Web Conference (ISWC) Demo Session.

Aini Rakhmawati, N., Umbrich, J., Karnstedt, M., Hasnain, A., and Hausenblas, M. (2013). Querying over Federated SPARQL Endpoints —A State of the Art Survey. *ArXiv e-prints*.

Akar, Z., Hala, T. G., Ekinci, E. E., and Dikenelli, O. (2012). Querying the web of interlinked datasets using void descriptions. In *Linked Data on the Web (LDOW2012)*.

Alexander, K. and Hausenblas, M. (2009). Describing linked datasets - on the design and usage of void, the vocabulary of interlinked datasets. In *In Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09*.

Arias, M., Fernández, J. D., Martínez-Prieto, M. A., and de la Fuente, P. (2011). An Empirical Study of Real-World SPARQL Queries. *ArXiv e-prints*.

Basca, C. and Bernstein, A. (2010). *Avalanche: Putting the Spirit of the Web back into Semantic Web Querying*.

Bechhofer, S., Buchan, I., De Roure, D., Missier, P., Ainsworth, J., Bhagat, J., Couch, P., Cruickshank, D., Delderfield, M., Dunlop, I., et al. (2011). Why linked data is not enough for scientists. *Future Generation Computer Systems*.

Beckett, D. (2014). Rdf 1.1 n-triples a line-based syntax for an rdf graph. W3C Recommendation. `http://www.w3.org/TR/n-triples/`.

Beckett, D., Berners-Lee, T., and Carothers, G. (2014). Rdf 1.1 turtle terse rdf triple language. W3C Recommendation. `http://www.w3.org/TR/turtle/`.

Berners-Lee, T. (1998). Notation 3 – Ideas about Web architecture. W3C Design Issues. `http://www.w3.org/DesignIssues/Notation3.html`.

Berners-Lee, T. (2006). Linked data. *W3C Design Issues*.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, 284(5):34–43.

Bizer, C. and Schultz, A. (2009). The berlin sparql benchmark. *International Journal On Semantic Web and Information Systems*.

Boncz, P. (2013). Ldbc: Benchmarks for graph and rdf data management. In *Proceedings of the 17th International Database Engineering & Applications Symposium*, IDEAS '13, pages 1–2, New York, NY, USA. ACM.

Cheung, K.-H., Frost, H. R., Marshall, M. S., Prud'hommeaux, E., Samwald, M., Zhao, J., and Paschke, A. (2009). A journey to semantic web query federation in the life sciences. *BMC Bioinformatics*, 10(S-10):10.

d'Aquin, M., Baldassarre, C., Gridinoc, L., Angeletou, S., Sabou, M., and Motta, E. (2007). Characterizing knowledge on the semantic web with watson. In *EON*, pages 1–10.

Deshpande, A., Ives, Z., and Raman, V. (2007). Adaptive query processing. *Found. Trends databases*, 1(1):1–140.

DeWitt, D. J. (1993). The wisconsin benchmark: Past, present, and future. In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*.

Dixit, K. M. (1991). The SPEC benchmarks. *Parallel Computing*, 17(10-11):1195–1209.

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

Gandon, F. and Schreiber, G. (2014). RDF 1.1 XML Syntax. W3C Recommendation. `http://www.w3.org/TR/rdf-syntax-grammar/`.

Gonzälez-Beltrn, A., Tagger, B., and Finkelstein, A. (2012). Federated ontology-based queries over cancer data. *BMC Bioinformatics*, 13(1):1–24.

Görlitz, O. and Staab, S. (2011). SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Proceedings of the 2nd International Workshop on Consuming Linked Data*, Bonn, Germany.

Görlitz, O., Thimm, M., and Staab, S. (2012). Splodge: Systematic generation of sparql benchmark queries for linked open data. In *International Semantic Web Conference (1)*, pages 116–132.

Gray, J. (1992). *Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Groth, P., Gibson, A., and Velterop, J. (2010). The anatomy of a nanopublication. *Inf. Serv. Use*, 30(1-2):51–56.

Guo, Y., Pan, Z., and Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158 – 182. Selcted Papers from the International Semantic Web Conference, 2004 - ISWC, 2004.

Guo, Y., Qasem, A., Pan, Z., and Heflin, J. (2007). A requirements driven framework for benchmarking semantic web knowledge base systems. *IEEE Trans. on Knowl. and Data Eng.*, 19(2):297–309.

Haas, L. M., Kossmann, D., Wimmers, E. L., and Yang, J. (1997). Optimizing queries across diverse data sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 276–285, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Haase, P., Mathäß, T., and Ziller, M. (2010). An evaluation of approaches to federated query processing over linked data. In *Proceedings of the 6th International Conference on Semantic Systems*, I-SEMANTICS '10, pages 5:1–5:9, New York, NY, USA. ACM.

Harland, L. (2012). Open phacts: A semantic knowledge infrastructure for public and commercial drug discovery research. In ten Teije, A., Vlker, J., Handschuh, S., Stuckenschmidt, H., d'Aquin, M., Nikolov, A., Aussenac-Gilles, N., and Hernandez, N., editors, *EKAW*, volume 7603 of *Lecture Notes in Computer Science*, pages 1–7. Springer.

Harris, S. and Seaborne (eds), A. (2013). SPARQL 1.1 query language. Working draft, W3C.

Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.-U., and Umbrich, J. (2010). Data summaries for on-demand queries over linked data. pages 411–420.

Hartig, O. (2011). Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *Proceedings of the 8th extended semantic web conference on The semantic web: research and applications - Volume Part I*, ESWC'11, pages 154–169, Berlin, Heidelberg. Springer-Verlag.

Hasnain, A., Fox, R., Decker, S., and Deus, H. F. (2012). Cataloguing and Linking Life Sciences LOD Cloud. In *18th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2012), OEDW 2012*.

Hausenblas, M. (2009). Exploiting linked data to build web applications. *Internet Computing, IEEE*, 13(4):68–73.

Hausenblas, M., Halb, W., Raimond, Y., Feigenbaum, L., and Ayers, D. (2009). Scovo: Using statistics on the web of data. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC 2009 Heraklion, pages 708–722, Berlin, Heidelberg. Springer-Verlag.

Hernandez, T. and Kambhampati, S. (2004). Integration of biological sources: current systems and challenges ahead. *SIGMOD Rec.*, 33(3):51–60.

Hose, K., Karnstedt, M., Koch, A., Sattler, K.-U., and Zinn, D. (2007). Processing rank-aware queries in p2p systems. In *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, DBISP2P'05/06, pages 171–178, Berlin, Heidelberg. Springer-Verlag.

Hose, K. and Schenkel, R. (2012). Towards benefit-based rdf source selection for sparql queries. In *Proceedings of the 4th International Workshop on Semantic Web Information Management*, SWIM '12, pages 2:1–2:8, New York, NY, USA. ACM.

Hose, K., Schenkel, R., Theobald, M., and Weikum, G. (2011). Database foundations for scalable rdf processing. In Polleres, A., dAmato, C., Arenas, M., Handschuh, S., Kroner, P., Ossowski, S., and Patel-Schneider, P., editors, *Reasoning Web. Semantic Technologies for the Web of Data*, volume 6848 of *Lecture Notes in Computer Science*, pages 202–249. Springer Berlin / Heidelberg.

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

Iqbal, A. and Hausenblas, M. (2013). Interlinking developer identities within and across open source projects: The linked data approach. *ISRN Software Engineering*.

Jain, P., Hitzler, P., Sheth, A. P., Verma, K., and Yeh, P. Z. (2010). Ontology alignment for linked open data. In *ISWC 2010*, pages 402–417. Springer.

Jentzsch, A., Hassanzadeh, O., Bizer, C., Andersson, B., and Stephens, S. (2009). Enabling tailored therapeutics with linked data. In *Proceedings of the 2nd Workshop on Linked Data on the Web (LDOW2009)*.

Jentzsch, A., Isele, R., and Bizer, C. (2010). Silk - generating rdf links while publishing or consuming linked data. In *Poster at the International Semantic Web Conference (ISWC2010), Shanghai*.

Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392.

Kikuchi, S., Sachdeva, S., Bhalla, S., Lynden, S., Kojima, I., Matono, A., and Tanimura, Y. (2010). Adaptive integration of distributed semantic web data. *Databases in Networked Information Systems*, 5999:174–193.

Kjernsmo, K. and Tyssedal, J. (2013). Introducing statistical design of experiments to sparql endpoint evaluation. In Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J., Aroyo, L., Noy, N., Welty, C., and Janowicz, K., editors, *The Semantic Web ISWC 2013*, volume 8219 of *Lecture Notes in Computer Science*, pages 360–375. Springer Berlin Heidelberg.

Langegger, A. and Woss, W. (2009). Rdfstats - an extensible rdf statistics generator and library. In *Proceedings of the 2009 20th International Workshop on Database and Expert Systems Application*, DEXA '09, pages 79–83, Washington, DC, USA. IEEE Computer Society.

Langegger, A., Wöß, W., and Blöchl, M. (2008). A semantic web middleware for virtual data integration on the web. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 493–507, Berlin, Heidelberg. Springer-Verlag.

Millard, I., Glaser, H., Salvadores, M., and Shadbolt, N. (2010). Consuming multiple linked data sources: Challenges and experiences. In *COLD*.

Montoya, G., Vidal, M.-E., and Acosta, M. (2012a). Defender: a decomposer for queries against federations of endpoints. 9th Extended Semantic Web Conference (ESWC) Demo Session.

Montoya, G., Vidal, M.-E., and Acosta, M. (2012b). A heuristic-based approach for planning federated sparql queries. In *Proceedings of the 3rd Consuming Linked Data Workshop (COLD 2012)*.

Montoya, G., Vidal, M.-E., Corcho, Ó., Ruckhaus, E., and Aranda, C. B. (2012c). Benchmarking federated sparql query engines: Are existing testbeds enough? In *International Semantic Web Conference (2)*, pages 313–324.

Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmér, M., and Risch, T. (2002). Edutella: A p2p networking infrastructure based on rdf. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 604–615, New York, NY, USA. ACM.

Neumann, T. and Moerkotte, G. (2011). Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 984 –994.

Ngomo, A.-c. N. and Auer, S. (2011). Limes - a time-efficient approach for large-scale link discovery on the web of data. In *Proceedings of IJCAI*, volume 15, pages 2312–2317.

Obermeier, P. and Nixon, L. (2011). A Cost Model for Querying Distributed RDF-Repositories with SPARQL. In *Proceedings of the Workshop on Advancing Reasoning on the Web: Scalability and Commonsense. CEUR Workshop Proceedings*, volume 350. Citeseer.

Peralta, V. (2006). Data freshness and data accuracy: A state of the art. Technical report, Universidad de la Republica.

Polleres, A. (2010). Semantic web technologies: From theory to standards. In *21st National Conference on Artificial Intelligence and Cognitive Science, NUI Galway*.

Prasser, F., Kemper, A., and Kuhn, K. A. (2012). Efficient distributed query processing

for autonomous rdf databases. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 372–383, New York, NY, USA. ACM.

Quackenbush, J. (2006). Standardizing the standards. *Molecular systems biology*, 2(1).

Quilitz, B. and Leser, U. (2008). Querying distributed rdf data sources with sparql. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 524–538, Berlin, Heidelberg. Springer-Verlag.

Raimond, Y., Sutton, C., and Sandler, M. B. (2009). Interlinking music-related data on the web. *IEEE MultiMedia*, 16(2):52–63.

Rakhmawati, N. and Umbrich, Jürgen and, M. (2013). A comparison of federation over sparql endpoints frameworks. In *Knowledge Engineering and the Semantic Web*, pages 132–146. Springer Berlin Heidelberg.

Rakhmawati, N. A. (2013a). An holistic evaluation of federated sparql query engines. In *Open Access Journal of Information Systems (OAJIS)*. ISICO.

Rakhmawati, N. A. (2013b). How interlinks influence federated over sparql endpoints. *International Journal of Internet and Distributed Systems*, 1(1):1–8.

Rakhmawati, N. A., Hasnain, A., Beyan, O., Iqbal, A., and Decker, S. (2014a). The cost and benefit of exploiting links in federated sparql query. In *WEBIST 2014*. SciTePress.

Rakhmawati, N. A. and Hausenblas, M. (2012). On the impact of data distribution in federated sparql queries. In *Semantic Computing (ICSC), 2012 IEEE Sixth International Conference on Semantic Computing*, pages 255 –260.

Rakhmawati, N. A., Karnstedt, M., Hausenblas, M., and Decker, S. (2014b). On metrics for measuring fragmentation of federation over sparql endpoints. In *WEBIST 2014*. SciTePress.

Rakhmawati, N. A., Saleem, M., Lalithsena, S., and Decker, S. (2014c). Qfed: Query set for federated sparql query benchmark. In *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '14, pages 207–211, New York, NY, USA. ACM.

Rakhmawati, N. A., Umbrich, J., Karnstedt, M., Hasnain, A., and Hausenblas, M. (2013). Querying over federated sparql endpoints - a state of the art survey. *CoRR*, abs/1306.1723.

Rohloff, K. and Schantz, R. E. (2010). High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA '10, pages 4:1–4:5, New York, NY, USA. ACM.

Saleem, M., Kamdar, M., Iqbal, A., Sampath, S., Deus, H., and Ngonga Ngomo, A.-C. (2014). Big linked cancer data: Integrating linked tcga and pubmed. *Web semantics: Science, Services and Agents on the World Wide Web*.

Saleem, M., Kamdar, M. R., Iqbal, A., Sampath, S., Deus, H. F., and Ngonga, A.-C. (2013a). Fostering serendipity through big linked data. In *Semantic Web Challenge at ISWC2013*.

Saleem, M., Ngomo, A.-C. N., Parreira, J. X., Deus, H. F., and Hauswirth, M. (2013b). Daw: Duplicate-aware federated query processing over the web of data. In *The Semantic Web–ISWC 2013*, pages 574–590. Springer.

Saleem, M., Padmanabhuni, S. S., Ngomo, A.-C. N., Almeida, J. S., Decker, S., and Deus, H. F. (2013c). Linked cancer genome atlas database. In *Proceedings of the 9th International Conference on Semantic Systems*, pages 129–134. ACM.

Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., and Tran, T. (2011a). Fedbench: A benchmark suite for federated semantic data query processing. In *International Semantic Web Conference (1)*, pages 585–600.

Schmidt, M., Grlitz, O., Haase, P., Ladwig, G., Schwarte, A., and Tran, T. (2011b). Fedbench: A benchmark suite for federated semantic data query processing. In Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N. F., and Blomqvist, E., editors, *International Semantic Web Conference (1)*, volume 7031 of *Lecture Notes in Computer Science*, pages 585–600. Springer.

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

Schultz, A., Matteini, A., Isele, R., Mendes, P. N., Bizer, C., and Becker, C. (2012). LDIF - A Framework for Large-Scale Linked Data Integration. In *21st International World Wide Web Conference (WWW2012), Developers Track*.

Schwarte, A., Haase, P., Hoose, K., Schenkel, R., and Schmidt, M. (2011). Fedx: A federation layer for distributed query processing on linked open data. In *ESWC*.

Schwarte, A., Haase, P., Schmidt, M., Hose, K., and Schenkel, R. (2012). An experience report of large scale federations. *CoRR*, abs/1210.5403.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. (1979). Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA. ACM.

Stankovic, M., Wagner, C., Jovanovic, J., and Laublet, P. (2010). Looking for experts? what can linked data do for you? In *LDOW*.

Stocker, M. and Seaborne, A. (2007). Arqo: The architecture for an arq static query optimizer.

Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., and Reynolds, D. (2008). Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA. ACM.

Tummarello, G., Delbru, R., and Oren, E. (2007). Sindice.com: weaving the open linked data. In *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, ISWC'07/ASWC'07, pages 552–565, Berlin, Heidelberg. Springer-Verlag.

Umbrich, J., Decker, S., Hausenblas, M., Polleres, A., and Hogan, A. (2010). Towards dataset dynamics: Change frequency of linked open data sources.

Umbrich, J., Hogan, A., Polleres, A., and Decker, S. (2012a). Improving the recall of live linked data querying through reasoning. In *Web Reasoning and Rule Systems*, pages 188–204. Springer.

Umbrich, J., Karnstedt, M., Hogan, A., and Parreira, J. (2012b). Hybrid sparql queries: Fresh vs. fast results. In Cudr-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J., Hendler, J., Schreiber, G., Bernstein, A., and Blomqvist, E., editors, *The Semantic Web ISWC 2012*, Lecture Notes in Computer Science, pages 608–624. Springer Berlin Heidelberg.

Umbrich, J., Karnstedt, M., Hogan, A., and Parreira, J. X. (2012c). Freshening up while

staying fast: Towards hybrid sparql queries. In *Knowledge Engineering and Knowledge Management*, pages 164–174. Springer.

Urhan, T. and Franklin, M. J. (2000). XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2):27–33.

van Rijsbergen, C. (1979). *Information Retrieval. 1979*. Butterworth.

Vidal, M.-E., Ruckhaus, E., Lampo, T., Martnez, A., Sierra, J., and Polleres, A. (2010). Efficiently joining group patterns in sparql queries. In Aroyo, L., Antoniou, G., Hyvnen, E., Teije, A., Stuckenschmidt, H., Cabral, L., and Tudorache, T., editors, *The Semantic Web: Research and Applications*, volume 6088 of *Lecture Notes in Computer Science*, pages 228–242. Springer Berlin Heidelberg.

Wang, X., Tiropanis, T., and Davis, H. (2011a). Querying the web of data with graph theory-based techniques.

Wang, X., Tiropanis, T., and Davis, H. C. (2011b). Evaluating graph traversal algorithms for distributed sparql query optimization. In Pan, J. Z., Chen, H., Kim, H.-G., Li, J., Wu, Z., Horrocks, I., Mizoguchi, R., and Wu, Z., editors, *JIST*, volume 7185 of *Lecture Notes in Computer Science*, pages 210–225. Springer.

Wilkinson, K. (2006). Jena property table implementation. In *In SSWS*.

Zemánek, J. and Schenk, S. (2008). Optimizing sparql queries over disparate rdf data sources through distributed semi-joins. In *International Semantic Web Conference (Posters & Demos)*.

Appendix A

Prefixes

Herein, we enumerate the prefixes used throughout this thesis to abbreviate URIs.

Prefix	URI
dailymed:	http://wifo5-03.informatik.uni-mannheim.de/dailymed/resource/dailymed/
dailymeddrug:	http://wifo5-03.informatik.uni-mannheim.de/dailymed/resource/drugs/
dailymeding:	http://wifo5-03.informatik.uni-mannheim.de/dailymed/resource/ingredients/
dailymedorg:	http://wifo5-03.informatik.uni-mannheim.de/dailymed/resource/organizations/
dbpedia:	http://dbpedia.org/resource/
disease:	http://wifo5-03.informatik.uni-mannheim.de/diseasome/resource/diseases/
diseasome:	http://wifo5-03.informatik.uni-mannheim.de/diseasome/resource/diseasome
drugbank:	http://wifo5-03.informatik.uni-mannheim.de/drugbank/resource/drugbank/
drugbankdrug:	http://wifo5-03.informatik.uni-mannheim.de/drugbank/resource/drugs/
foaf:	http://xmlns.com/foaf/0.1/
owl:	http://www.w3.org/2002/07/owl#
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
sider:	http://wifo5-03.informatik.uni-mannheim.de/sider/resource/sider/
siderdrug:	http://wifo5-03.informatik.uni-mannheim.de/sider/resource/sider/drugs/
xsd:	http://www.w3.org/2001/XMLSchema#

Table A.1: Used prefixes

Appendix B

Query Set

B.1 Dailymed Queries

The following queries are used in the evaluations in Chapter 4 and 7.

Listing B.1: 16 Dailymed Queries

```
select *
{
    ?drug a dailymed:drugs .
    ?drug dailymed:activeingredient ?active .
}


select *
{
    ?organization a dailymed:organization .
    ?organization dailymed:producesdrug ?drug .
}


select *
{
    dailymeddrug:1008 ?p ?o .
}
```

```
select *
{
    ?drug ?p dailymeding:Phenytoin_Sodium
}
```

```
select *
{
    ?thing rdfs:label ?label
}
```

```
select *
{
    ?drug a dailymed:drugs .
    ?drug rdfs:label ?label .
}
```

```
select *
{
    ?org a dailymed:organization .
    ?org rdfs:label ?label .
}
```

```
select *
{
    ?drug a dailymed:drugs .
    ?drug dailymed:routeofadministration <http://www4.wiwiss.fu-berlin
        .de/dailymed/resource/routeofadministration/intramuscular> .
    ?drug dailymed:representedorganization <http://www4.wiwiss.fu-
        berlin.de/dailymed/resource/organization/hospira\%2c_inc.> .
}
```

```
select *
{
    ?drug a dailymed:drugs .
    ?drug dailymed:routeofadministration <http://www4.wiwiss.fu-berlin
        .de/dailymed/resource/routeofadministration/intramuscular> .
    ?drug rdfs:label ?label .
}
```

```
select *
{
    ?drug a dailymed:drugs .
    ?drug dailymed:possiblediseasetarget disease:1883 .
    ?drug rdfs:label ?label .
}
```

```
select *
{
    ?drug a dailymed:drugs .
    ?drug dailymed:activeingredient ?active .
    ?active rdfs:label ?label .
}
```

```
select *
{
    ?drug a dailymed:drugs .
    ?drug dailymed:representedorganization ?org .
    ?org rdfs:label ?label .
}
```

```
select *
{
    dailymeddrug:1905 dailymed:representedorganization ?org .
    ?org dailymed:producesdrug ?otherdrug .
```

```
}


select *
{
    dailymeddrug:3369 dailymed:representedorganization ?org .
    ?org dailymed:producesdrug ?otherdrug .
}


select ?drugname ?orgname
{
    ?drug dailymed:representedorganization ?org .
    ?drug rdfs:label ?drugname .
    ?org rdfs:label ?orgname .
}


select ?drugname ?ingridient
{
    ?drug dailymed:activeingredient ?active .
    ?drug rdfs:label ?drugname .
    ?active rdfs:label ?ingridient .
}
```

B.2 Link and No Link Queries

The following queries are used in the evaluations in Chapter 5.

Listing B.2: Link and No Link Queries

```
select distinct ?siderdrug ?drugname
{
    drugbankdrug:DB00316 rdfs:label ?drugname .
    ?siderdrug a sider:drugs .
    ?siderdrug rdfs:label ?drugname .
```

```
}

select distinct ?siderdrug ?drugname
{
        drugbankdrug:DB00316 owl:sameAs ?siderdrug .
        ?siderdrug a sider:drugs .
        ?siderdrug rdfs:label ?drugname .
}

select distinct ?siderdrug ?drugname
{
        drugbankdrug:DB00984 rdfs:label ?drugname .
        ?siderdrug a sider:drugs .
        ?siderdrug rdfs:label ?drugname .
}

select distinct ?siderdrug ?drugname
{
        drugbankdrug:DB00984 owl:sameAs ?siderdrug .
        ?siderdrug a sider:drugs .
        ?siderdrug rdfs:label ?drugname .
}

select distinct ?anydrug ?drugname
{
        siderdrug:951 rdfs:label ?drugname .
        ?anydrug a drugbank:drugs> .
        ?anydrug rdfs:label ?drugname .
}

select distinct ?anydrug ?drugname
{
```

```
        siderdrug:951 owl:sameAs ?anydrug .
        ?anydrug a drugbank:drugs> .
        ?anydrug rdfs:label ?drugname .
}


select distinct ?anydrug ?drugname
{
        siderdrug:727 rdfs:label ?drugname .
        ?anydrug a dailymed:drugs .
        ?anydrug dailymed:name ?drugname .
}


select distinct ?anydrug ?drugname
{
        siderdrug:727 owl:sameAs ?anydrug .
        ?anydrug a drugbank:drugs .
        ?anydrug rdfs:label ?drugname .
}


select distinct ?anydrug ?drugname
{
        dailymeddrug:60 dailymed:genericMedicine ?drugname .
        ?anydrug a drugbank:drugs .
        ?anydrug rdfs:label ?drugname .
}


select distinct ?anydrug ?drugname
{
        dailymeddrug:60 dailymed:genericDrug ?anydrug .
        ?anydrug a drugbank:drugs .
        ?anydrug drugbank:brandName ?drugname .
}
```

```
select distinct ?anydrug ?drugname
{
        dailymeddrug:3460 dailymedname ?drugname .
        ?anydrug a drugbank:drugs .
        ?anydrug rdfs:label ?drugname .
}


select distinct ?anydrug ?drugname
{
        dailymeddrug:3460 dailymed:genericDrug ?anydrug .
        ?anydrug a drugbank:drugs .
        ?anydrug rdfs:label ?drugname .
}
```

B.3 QFed Queries

There are many queries that are generated by QFed. We will show some of those queries as follows:

Listing B.3: Example of a subject-object join pattern query that joins a class to entities and uses SERVICE keywords

```
select * {
        service<http://localhost:8001/dailymed/query> {
                ?s1 a dailymed:drugs .
                ?s1 dailymed:possibleDiseaseTarget ?s2 . }
        service<http://localhost:8002/disease/query>  {
                ?s2 diseasome:associatedGene ?URI .
                ?s2 diseasome:classDegree ?LITERAL .
                }
 }
```

Listing B.4: Example of a subject-object join pattern query that joins a class to entities

```
select * {
```

```
        ?s1 a dailymed:drugs> .
        ?s1 dailymed:possibleDiseaseTarget ?s2 .
        ?s2 diseasome:associatedGene ?URI .
        ?s2 diseasome:classDegree ?LITERAL .
}
```

Listing B.5: Example of a subject-object join pattern query that joins a class to entities and uses an `OPTIONAL` keyword

```
select * {
        ?s1 a dailymed:drugs> .
         ?s1 dailymed:possibleDiseaseTarget ?s2 .
        ?s2 diseasome:associatedGene ?URI .
        OPTIONAL {
                ?s2 diseasome:classDegree ?LITERAL .
        } }
```

Listing B.6: Example of a subject-object join pattern query that joins a class to entities and uses a `FILTER` keyword

```
select * {
        ?s1 a dailymed:drugs> .
        ?s1 dailymed:possibleDiseaseTarget ?s2 .
        ?s2 diseasome:associatedGene ?URI .
        ?s2 diseasome:classDegree ?LITERAL .
        FILTER(?LITERAL >= 2) .
 }
```

Listing B.7: Example of a subject-object join pattern query that joins amongst entities and uses a big literal object

```
select * {
        dailymeddrug:1004 dailymed:genericDrug ?s2 .
        ?s2 drugbank:casRegistryNumber ?URI .
        ?s2 drugbank:absorption ?BIGLITERAL .
}
```

Listing B.8: Example of a object-object join pattern query that joins amongst entities and uses a big literal object

```
select * {
    dailymeding:Amcinonide rdfs:label ?o .
    ?s2 a <http://www4.wiwiss.fu-berlin.de/drugbank/vocab/resource/
        class/Offer> .
    ?s2 drugbank:genericName ?o .
    ?s2 drugbank:casRegistryNumber ?URI .
    ?s2 drugbank:absorption ?BIGLITERAL .
}
```

Listing B.9: Example of a hybrid join pattern query that joins amongst entities

```
select * {
    <http://www4.wiwiss.fu-berlin.de/diseasome/resource/genes/HMS>
        rdfs:label ?o .
    ?s2 a dailymed:drugs> .
    ?s2 dailymed:name ?o .
    ?s2 dailymed:activeIngredient ?URI .
    ?s2 dailymed:fullName ?LITERAL .
    ?s2 dailymed:genericDrug ?s3 .
    ?s3 drugbank:casRegistryNumber ?URI3 .
}
```

Listing B.10: Example of a subject-subject join pattern query

```
select * {
    ?s drugbank:interactionDrug1 ?URI .
    ?s drugbank:interactionDrug2 ?URI2 .
    ?s drugbank:text ?BIGLITERAL .
}
```

Appendix C

Dataset

Herein, we provide the statistical information about our dataset that are used in our evaluation in Chapter 7.

Dataset	Partition 1	Partition 2	Partition 3
CD	147245	11212	5773
GD	26767	58657	97380
TD	54759	54759	54758
TD2	10	10	10
TD3	54760	54756	54760
HD	54744	54744	54786
HD2	54749	54749	54779
PD	52510	56507	55259
ED	119261	34502	10513

Table C.1: List of Number of Triples in each dataset.

Dataset	Partition 1	Partition 2	Partition 3
CD	2	1	1
GD	5	4	5
TD	4	0	2
TD2	10	10	10
TD3	5	8	0
HD	2	2	6
HD2	2	3	4
PD	0	0	6
ED	4	3	4

Table C.2: List of Number of Classes in each dataset.

Dataset	Partition 1	Partition 2	Partition 3
CD	26	3	4
GD	27	26	28
TD	13	3	17
TD2	10	10	10
TD3	21	17	3
HD	26	26	28
HD2	27	26	28
PD	14	9	5
ED	27	26	5

Table C.3: List of Number of Properties in each dataset.

Dataset	Partition 1	Partition 2	Partition 3
CD	4308	4066	711
GD	1893	4628	9090
TD	5058	4176	8882
TD2	10	10	10
TD3	5076	8328	4176
HD	1630	1580	5915
HD2	2836	4260	4692
PD	9085	4311	9125
ED	3533	3144	2540

Table C.4: List of Number of Entities in each dataset.

Appendix D

The accuracy results

We carried out an experiment in Chapter 5 which investigates the impact of the links on the accuracy of federated SPARQL query results. The following are the queries along with its results:

Listing D.1: Query asking for a list of Branded Drug and its Generic Drug based on its inactive ingredients

```
select ?brandname ?genericname {
        ?drug a dailymed:drugs .
        ?drug rdfs:label ?brandname .
        ?drug dailymed:inactiveIngredient ?ingredients .
        ?ingredients rdfs:label ?genericname .
        ?anydrug a drugbank:drugs .
        ?anydrug rdfs:label ?genericname .
}
limit 10
```

The query results:

Listing D.2: Query asking for a list of Branded Drug and its Generic Drug based on the active ingredients

```
select ?brandname ?genericname {
        ?drug a dailymed:drugs .
        ?drug rdfs:label ?brandname .
        ?drug dailymed:activeIngredient ?ingredients .
        ?ingredients rdfs:label ?genericname .
```

Branded Drug	Generic Drug	Validation
Evista (Tablet)	Lactose	False
Lisinopril (Tablet)	Mannitol	False
Triamcinolone Acetonide (Cream)	Lactic Acid	False
Aminonide (Cream)	Lactic Acid	False
Fluocinolone Acetonide (Cream)	Citric Acid	False
Acetaminophen and Codeine Phosphate (Solution)	Sucrose	False
Capoten (Tablet)	Lactose	False
Macrotec	Acetic Acid	False
Uniretic	Lactose	False
Sodium Acetate	Acetic Acid	False

```
        ?anydrug a drugbank:drugs .
        ?anydrug rdfs:label ?genericname .
}
limit 10
```

The query results:

Branded Drug	Generic Drug	Validation
Gabapentin (Capsule)	Gabapentin	True
ZESTORETIC (Tablet)	Lisinopril	False
Lisinopril (Tablet)	Lisinopril	True
Ketoconazole (Tablet)	Ketoconazole	True
Zazole (Cream)	Terconazole	True
Lovastatin (Tablet)	Lovastatin	True
Capoten (Tablet)	Captopril	True
Lovastatin (Tablet)	Lovastatin	True
Aminonide (Cream)	Amcinonide	True
Temazepam (Capsule)	Temazepam	True

Listing D.3: Query asking for a list of Branded Drug and its Generic Drug based on active Moiety

```
select ?brandname ?genericname {
        ?drug a dailymed:drugs .
        ?drug rdfs:label ?brandname .
        ?drug dailymed:activeMoiety ?ingredients .
        ?ingredients rdfs:label ?genericname .
        ?anydrug a drugbank:drugs .
```

```
    ?anydrug rdfs:label ?genericname .
}
limit 10
```

The query results:

Branded Drug	Generic Drug	Validation
Evista (Tablet)	Raloxifene	True
Gabapentin (Capsule)	Gabapentin	True
Vasocidin (Solution)	Prednisolone	False
Pergolide Mesylate (Tablet)	Pergolide	True
Galantamine (Tablet)	Galantamine	True
Penicillin V Potassium (Tablet)	Penicillin V	True
ZESTORETIC (Tablet)	Lisinopril	False
"SEROQUEL (Tablet & Coated)"	Quetiapine	True
Acebutolol Hydrochloride (Capsule)	Acebutolol	True
Bacitracin (Ointment)	Bacitracin	True

Listing D.4: Query asking for a list of Branded Drug and its Generic Drug by using `dailymed:genericDrug` link

```
select ?brandname ?genericname {
    ?drug a dailymed:drugs .
    ?drug rdfs:label ?brandname .
    ?drug dailymed:genericDrug ?anydrug .
    ?anydrug rdfs:label ?genericname .
}
limit 10
```

The query results:
To validate the query result, we ask two following healthcare experts:

1. *Arief Budi Santosa*
   Work: Kepanjen Hospital, Malang, East Java, Indonesia
   Education:

   - Cardiologist, Airlangga University, Surabaya, East Java, Indonesia (-present)
   - Medical Education, Hang Tuah University, Surabaya, East Java, Indonesia ()

Branded Drug	Generic Drug	Validation
Leflunomide (Tablet)	Leflunomide	True
"Arava (Tablet & Film Coated)"	Leflunomide	True
Zazole (Cream)	Terconazole	True
Terconazole (Suppository)	Terconazole	True
Zazole (Suppository)	Terconazole	True
"Lozol (Tablet & Film Coated)"	Indapamide	True
"ZOCOR (Tablet & Film Coated)"	Simvastatin	True
Halotestin (Tablet)	Fluoxymesterone	True
ANDROXY (Tablet)	Fluoxymesterone	True
COSMEGEN	Dactinomycin	True

2. *Aslihan Beyan*
   Work: Pharmacist in Turkish Ministry of Health - Health Research Department
   Education: Master Degree, Natural Products Chemistry and Pharmacognosy, Ankara University, Institute of Health Science (1996-2000)

List of Algorithms

8.1 Subject-object Join Pattern 120
8.2 Object-object Join Pattern 122
8.3 Subject-subject Join Pattern 123
8.4 Hybrid Join Pattern 124
8.5 Star Function 126
8.6 Star Function for Hybrid Join 127

List of Figures

1.1 Example data relation from the Drugbank and the Kegg datasets 3
1.2 Dependencies between metrics, datasets and queries 4
1.3 Heterogeneity in federated SPARQL endpoints framework 7
1.4 Example of query execution for searching students along with their courses . 11

2.1 Representation of a RDF statement. 17
2.2 Representation of RDF triples 17
2.3 Example of a SPARQL basic graph pattern (BGP) represented as a graph. . 20
2.4 Matching parts of the source graph in Figure 2.2 and BGP in Figure 2.3 . . 20
2.5 Example of Linked Data implementation on Diseasome and Drugbank datasets. 26
2.6 Centralised repository 27
2.7 P2P Systems 28
2.8 Link Traversal 28
2.9 Federation over single RDF repositories 29
2.10 Federation over SPARQL endpoints 29

3.1 Architecture of federation over SPARQL endpoints 33
3.2 The Relationship amongst entities in Drugbank, Disease, Dailymed and Sider datasets 34
3.3 Federated SPARQL query Process in a federated engine 38

4.1 The section of infrastructure for assessing query federation framework 58
4.2 The relationship between federated engine components and independent metrics. 58
4.3 Response Time (seconds). 65
4.4 Performance of the three federated engines measured by independent metrics 67
4.5 Performance of the three Federated Engines Measured By Semi-Independent Metrics 68

4.6 Combined Indepedent Metrics Results 69

5.1 The example of the hidden cost of links 72
5.2 A sample of a federation over SPARQL endpoints where some entities in SPARQL endpoint are interlinked with `owl:sameAs` 74
5.3 Average Number of Rows 81
5.4 Average Response Time (in log scale) 82
5.5 Average Number of Requests 82
5.6 Average Intermediate Results (in log scale) 83
5.7 Average Data Transmission (in log scale) 83
5.8 Average Data Sent and Received (Bytes) Versus the number of Links (log scale) 84

6.1 Example of METIS Partition applied 87

7.1 Example of a federated SPARQL query involving many datasets 98
7.2 The OCP values for Equation 7.1 101
7.3 The OCP values for Equation 7.2 102
7.4 The occurrences of classes and properties in the dataset 107
7.5 Spreading factor of the dataset 108
7.6 The distribution of number of triples amongst the dataset partitions 108
7.7 The distribution of coherence values amongst the dataset partitions 109
7.8 Pearson Correlation Vs β 110
7.9 Average Data Transfer Volume Vs the spreading factor of Datasets (order by the spreading factor value) 111
7.10 Average Data Transfer Volume Vs the Q-spreading factor of Datasets associated with the Queryset(order by the spreading factor value) 112

8.1 QFed Input Output Diagram 115
8.2 Dataset Example 116
8.3 Federated Query Templates 121
8.4 Average Volume Data Transmission on Fuseki Execution (in bytes) 136
8.5 FedX Execution Results For Threshold 2 137
8.6 FedX Execution Results For Threshold 3 137

List of Tables

3.1 The existing federated engines using the rewriter architecture approach. . . . 43
3.2 The existing federated engine applying the planner architecture approach. . . 47

4.1 The list of existing evaluations of a federated SPARQL query 56
4.2 Source-set Used in Evaluation 66

5.1 Statistic for the dataset used in the evaluation 78
5.2 The Accuracy of Query Results (Experiment Three) 84

6.1 Example of Vertically Partitioned Tables applied 88
6.2 Example of Clustered Property Table applied 89
6.3 Example of Property-class Table Applied 89

7.1 The combinations of distribution of properties 100
7.2 Query Characteristic 106
7.3 Dataset used in evaluation 107
7.4 Evaluation Results 110
7.5 Pearson Correlation Evaluation Results 110

8.1 Features and Capabilities of QFed, Lidaq and Splodge. S=Subject,O=Object 132
8.2 The list of Query-set Generation (S-S=Subject-Subject Join,O-O=Object-Object Join,S-O=Subject-Object Join,H=Hybrid Join) 134
8.3 The Number of BGPs for each queries in the query set 134
8.4 Statistical queries containing `FILTER`, `OPTIONAL` and big literal object value . 135

A.1 Used prefixes 156

C.1 List of Number of Triples in each dataset. 166
C.2 List of Number of Classes in each dataset. 167

C.3 List of Number of Properties in each dataset. 167
C.4 List of Number of Entities in each dataset. 167

Listings

2.1 Example of a SPARQL query for retrieving all triples relating to `diseasome:diseases` 19
2.2 Example of a SPARQL query for retrieving a list of diseases and its drugs . . 19
2.3 Example of a SPARQL query for retrieving a list of diseases and its drugs by using the `OPTIONAL` keyword 21
2.4 Example of a SPARQL query for retrieving a list of diseases information by using an `UNION` keyword 21
2.5 Example of a SPARQL `CONSTRUCT` query for retrieving a list of diseases with size 1 22
2.6 Example of a SPARQL `ASK` query to check a list of diseases with size 1 . . . 22
2.7 Example of a SPARQL `DESCRIBE` query for retrieving all triples relating to `disease:383` 22
2.8 Example of a SPARQL query for retrieving all diseases which disease's size is greater than one by using the `FILTER` 23
2.9 Example of a SPARQL algebra for query in Listing 2.2 23
2.10 Example of a SPARQL algebra for query in Listing 2.3 24
2.11 Example of a SPARQL algebra for query in Listing 2.4 24
2.12 Example of a SPARQL algebrafor query in Listing 2.8 24
3.1 Example of a federated SPARQL query in SPARQL 1.0 31
3.2 Example of the same federated SPARQL query in SPARQL 1.1 with the `SERVICE` keyword 31
3.3 Example of a federated SPARQL query without the SPARQL endpoint specified 33
3.4 Example of a federated SPARQL query using the Popular Predicate 35
3.5 Example of a federated SPARQL query Using a variable in the predicate position 35
3.6 Example of a federated SPARQL query using a `UNION` operator 36
3.7 Example of a federated SPARQL query using label comparison 36
3.8 Example of a federated SPARQL query using URIs comparison 36

3.9 Example of a federated SPARQL query using links 37
3.10 Example of an ASK query to select the most relevant source 39
4.1 List of `ASK` queries sent to determine relevant sources for query in Listing 3.3 61
4.2 List of sub queries sent to SPARQL endpoints to accomplish query in Listing 3.3 62
5.1 Drugbank Sample Dataset 76
6.1 Dailymed Sample Triples 87
8.1 Example of a Subject-Object join pattern Query 128
8.2 Example of an Object-Object join pattern query 128
8.3 Example of a Subject-Subject join pattern query 129
8.4 Example of a hybrid join pattern query 130
8.5 Example of a federated SPARQL query using the FILTER and `OPTIONAL` Keywords 130
8.6 Example of a federated SPARQL Query using the SERVICE keyword 130
B.1 16 Dailymed Queries 157
B.2 Link and No Link Queries 160
B.3 Example of a subject-object join pattern query that joins a class to entities and uses SERVICE keywords 163
B.4 Example of a subject-object join pattern query that joins a class to entities . 163
B.5 Example of a subject-object join pattern query that joins a class to entities and uses an `OPTIONAL` keyword 164
B.6 Example of a subject-object join pattern query that joins a class to entities and uses a `FILTER` keyword 164
B.7 Example of a subject-object join pattern query that joins amongst entities and uses a big literal object 164
B.8 Example of a object-object join pattern query that joins amongst entities and uses a big literal object 164
B.9 Example of a hybrid join pattern query that joins amongst entities 165
B.10 Example of a subject-subject join pattern query 165
D.1 Query asking for a list of Branded Drug and its Generic Drug based on its inactive ingredients 168
D.2 Query asking for a list of Branded Drug and its Generic Drug based on the active ingredients 168
D.3 Query asking for a list of Branded Drug and its Generic Drug based on active Moiety 169

D.4 Query asking for a list of Branded Drug and its Generic Drug by using `dailymed:genericDrug` link 170

"I have made this letter longer than usual, only because I have not had the time to make it shorter."

—Blaise Pascal