EFFICIENT SOURCE SELECTION FOR SPARQL ENDPOINT QUERY FEDERATION

Der Fakultät für Mathematik und Informatik
der Universität Leipzig eingereichte

DISSERTATION

zur Erlangung des akademischen Grades

DOCTOR RERUM NATURALIS
(Dr. rer. nat.)

im Fachgebiet Informatik vorgelegt von

M.Sc Muhammad Saleem

geboren am 03.03.1984 in Bannu, Pakistan

Leipzig, den 21. Mai 2016

[21. Mai 2016 at 23:41 – classicthesis version 4.1]

BIBLIOGRAPHIC DATA

TITLE:
Efficient Source Selection for SPARQL Endpoint Query Federation

AUTHOR:
Muhammad Saleem

STATISTICAL INFORMATION:
208 pages, 71 Figures, 46 tables, 21 listings

SUPERVISORS:
Prof. Dr.-Ing. habil. Klaus-Peter Fähnrich
Dr. Axel-Cyrille Ngonga Ngomo

INSTITUTION:
Universität Leipzig, Fakultät für Mathematik und Informatik

TIME FRAME:
August 2012 - October 2015

ABSTRACT

The Web of Data has grown enormously over the last years. Currently, it comprises a large compendium of linked and distributed datasets from multiple domains. Due to the decentralised architecture of the Web of Data, several of these datasets contain complementary data. Running complex queries on this compendium thus often requires accessing data from different data sources within one query. The abundance of datasets and the need for running complex query has thus motivated a considerable body of work on SPARQL query federation systems, the dedicated means to access data distributed over the Web of Data.

This thesis addresses two key areas of federated SPARQL query processing: (1) efficient source selection, and (2) comprehensive SPARQL benchmarks to test and ranked federated SPARQL engines as well as triple stores.

Efficient Source Selection: Efficient source selection is one of the most important optimization steps in federated SPARQL query processing. An overestimation of query relevant data sources increases the network traffic, result in irrelevant intermediate results, and can significantly affect the overall query processing time. Previous works have focused on generating optimized query execution plans for fast result retrieval. However, devising source selection approaches beyond triple pattern-wise source selection has not received much attention. Similarly, only little attention has been paid to the effect of duplicated data on federated querying. This thesis presents HiBISCuS and TBSS, novel hypergraph-based source selection approaches, and DAW, a duplicate-aware source selection approach to federated querying over the Web of Data. Each of these approaches can be combined directly with existing SPARQL query federation engines to achieve the same recall while querying fewer data sources. We combined the three (HiBISCuS, DAW, and TBSS) source selections approaches with query rewriting to form a complete SPARQL query federation engine named Quetsal. Furthermore, we present TopFed, a Cancer Genome Atlas (TCGA) tailored federated query processing engine that exploits the data distribution to perform intelligent source selection while querying over large TCGA SPARQL endpoints. Finally, we address the issue of rights managements and privacy while accessing sensitive resources. To this end, we present SAFE: a global source selection approach that enables decentralised, policy-aware access to sensitive clinical information represented as distributed RDF Data Cubes.

Comprehensive SPARQL Benchmarks: Benchmarking is indispens-

able when aiming to assess technologies with respect to their suitability for given tasks. While several benchmarks and benchmark generation frameworks have been developed to evaluate federated SPARQL engines and triple stores, they mostly provide a one-fits-all solution to the benchmarking problem. This approach to benchmarking is however unsuitable to evaluate the performance of a triple store for a given application with particular requirements. The fitness of current SPARQL query federation approaches for real applications is difficult to evaluate with current benchmarks as current benchmarks are either synthetic or too small in size and complexity. Furthermore, state-of-the-art federated SPARQL benchmarks mostly focused on a single performance criterion, i.e., the overall query runtime. Thus, they cannot provide a fine-grained evaluation of the systems. We address these drawbacks by presenting FEASIBLE, an automatic approach for the generation of benchmarks out of the query history of applications, i.e., query logs and LargeRDFBench, a billion-triple benchmark for SPARQL query federation which encompasses real data as well as real queries pertaining to real bio-medical use cases.

Our evaluation results show that HiBISCuS, TBSS, TopFed, DAW, and SAFE all can significantly reduce the total number of sources selected and thus improve the overall query performance. In particular, TBSS is the first source selection approach to remain under 5% overall relevant sources overestimation. QUETSAL has reduced the number of sources selected (without losing recall), the source selection time as well as the overall query runtime as compared to state-of-the-art federation engines. The LargeRDFBench evaluation results suggests that the performance of current SPARQL query federation systems on simple queries does not reflect the systems' performance on more complex queries. Moreover, current federation systems seem unable to deal with many of the challenges that await them in the age of Big Data. Finally, the FEASIBLE's evaluation results shows that it generates better sample queries than the state-of-the-art. In addition, the better query selection and the larger set of query types used lead to triple store rankings which partly differ from the rankings generated by previous works.

PUBLICATIONS

This thesis is based on the following publications and proceedings.

AWARDS AND NOTABLE MENTIONS

- **Semantic Web Challenge (Big Data Track) Winner** at ISWC2013 for *Fostering Serendipity through Big Linked Data* [85]
- **I-Challenge (Linked Data Cup) Winner** at I-Semantics2013 for *Linked Cancer Genome Atlas Database* [89]
- **Best Paper Award** at CSHALS2014 for *Fragmenting the Genomic Wheel to Augment Discovery in Cancer Research* [47]

INTERNATIONAL JOURNALS, PEER-REVIEWED

- Semantic Web Journal, 2014: "*A Fine-Grained Evaluation of SPARQL Endpoint Federation Systems*" [84]
- Journal of Biomedical Semantics, 2014: "*TopFed: TCGA Tailored Federated Query Processing and Linking to LOD*" [90]
- Web Semantics: Science, Services and Agents on the World Wide Web, 2014: "*Big Linked Cancer Data: Integrating Linked TCGA and PubMed*" [81]

CONFERENCES, PEER-REVIEWED

- International Semantic Web Conference (ISWC), 2015: "*FEASIBLE: A Featured-Based SPARQL Benchmarks Generation Framework* " [78]
- International Semantic Web Conference (ISWC), 2015: "*LSQ: The Linked SPARQL Queries Dataset* " [76]
- Extended Semantic Web Conference (ESWC), 2014: "*HiBISCuS: Hypergraph-based Source Selection for SPARQL Endpoint Federation*" [87]
- International Semantic Web Conference (ISWC), 2013: "*DAW: Duplicate-AWare Federated Query Processing over the Web of Data*" [77]

- Semantic Web Applications and Tools for the Life Sciences (SWAT4LS), 2014: "*SAFE: Policy Aware SPARQL Query Federation Over RDF Data Cubes*" [52]
- International Conference on Information Integration and Web-based Applications & Services (iiWAS), 2014: "*Fed: Query Set For Federated SPARQL Query Benchmark*" [72]

UNDER-REVIEW

- To be submitted, 2015: "*QUETSAL: A Query Federation Suite for SPARQL*" [88]
- Journal of Web Semantics (JWS), 2015: "*LargeRDFBench: A Billion Triples Benchmark for SPARQL Query Federation* " [79]

ACKNOWLEDGMENTS

First of all I would like to thank my supervisors, Dr. Axel-Cyrille Ngonga Ngomo and Prof. Klaus-Peter Fähnrich, without whom I could not have started my Ph.D. at the Leipzig University.

Special thanks to my direct supervisor Dr. Axel-Cyrille Ngonga Ngomo, with whom I have started work on my Ph.D. proposal submitted to Deutscher Akademischer Austauschdienst (DAAD), in order to pursue my Ph.D. at the Agile Knowledge Engineering and Semantic Web (AKSW) group. He has continuously supported me throughout my Ph.D. work, giving advices and recommendations for further research steps. His comments and notes were very helpful for me particularly during the writing of the papers we published together. I would like to thank him also for proofreading this thesis and for his helpful feedback, which led to improving the quality of that thesis. Special thanks also to thank Prof. Sören Auer, whom I have first contacted asking for a vacancy to conduct my Ph.D. research at his research group. I am thankful to Dr. Jens Lehman for valuable discussions during my Ph.D. work

I would like also to thank Prof. Klaus-Peter Fähnrich, for the regular follow-up meetings he has managed in order to evaluate the performance of all Ph.D. students. During these meetings, he has proposed several directions for me and for other Ph.D. students as well, on how to deepen and extend our research points. I would like to thank all of my colleagues in the Semantic Abstraction (SIMBA) group, for providing me with their useful comments and guidelines, especially during the initial phase of my Ph.D..

I would like to dedicate this work to the souls of my parents, without whom I could not do anything in my life. With their help and support, I could take my first steps in my scientific career. Special thanks goes to all my family members and my friend Ahsan Rasheed.

CONTENTS

LISTE DER NOCH ZU ERLEDIGENDEN PUNKTE

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

The Web of Data is now a large compendium of interlinked data sets from multiple domains with large datasets [89] being added frequently [67]. Given the complexity of information needs on the Web, certain queries can only be answered by retrieving results contained across different data sources (short: sources). Thus, the optimization of engines that support this type of queries, called *federated query engines*, is of central importance to ensure the usability of the Web of Data in real-world applications.

Current federation engines however suffer of two main drawbacks: First, the selection of sources remains sub-optimal. In addition, the evaluation of the performance of federated engines remains a difficult process as the benchmarks available for this purpose are not comprehensive. After a brief explanation of the general steps involved in federated query processing, we present and discuss the need for efficient source selection and more comprehensive SPARQL benchmarks in more detail. This thesis' contributions are presented subsequently. Finally, a short introduction of each of the chapters is provided.

1.1 FEDERATED SPARQL QUERY PROCESSING

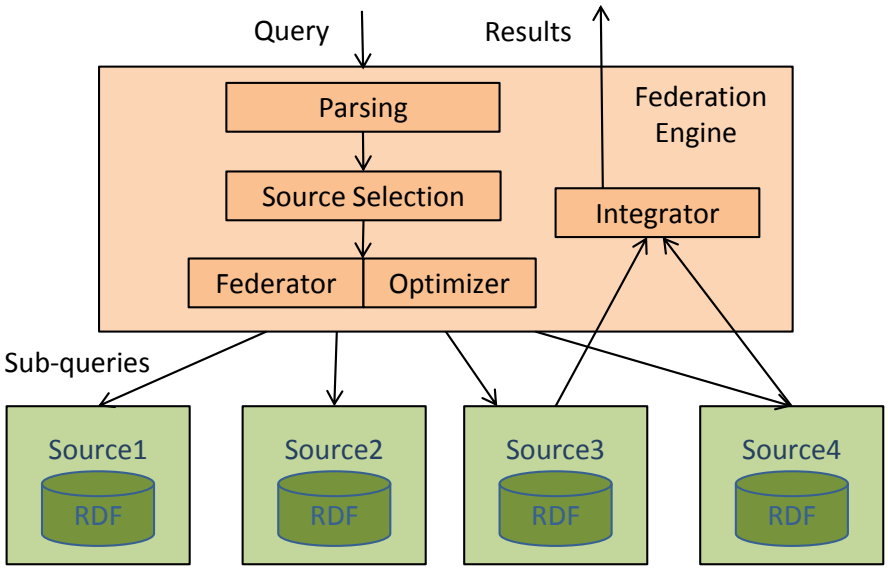


Figure 1: General Steps Involved in Federated SPARQL Query Processing.

Figure 1 shows the general steps involved in a federated SPARQL query processing. Given a SPARQL query, the first step is to parse

the query and get the individual triple patterns. The next step is the source selection; the goal of the source selection is to identify the set of relevant (also called capable) sources for the query. Using the source selection information, the federator divides the original query into multiple sub-queries. An optimized sub-query execution plan is generated by the optimizer and the sub-queries are forwarded to the corresponding data sources. The results of the sub-queries are then integrated by the integrator. The integrated results is finally returned to the agent that issued the query.

1.2 THE NEED FOR EFFICIENT SOURCE SELECTION

One of the important optimization steps in federated SPARQL query processing is the efficient selection of relevant sources for a query. To ensure that a recall of 100% is achieved, most SPARQL query federation approaches [27; 57; 70; 94; 100; 77] perform a *triple pattern-wise source selection* (TPWSS). The goal of TPWSS is to identify the set of relevant sources against individual triple patterns of a query [77]. However, it is possible that a relevant source does not *contribute* to the final result set of the complete query. This is because the results from a particular data source can be excluded after performing *joins* with the results of other triple patterns contained in the same query. An overestimation of such sources increases the network traffic by retrieving irrelevant intermediate results and can significantly affect the overall query processing time due to sub-optimal query execution plan selection. In the next paragraph, we present an example of such query from FedBench [91], a well-known benchmark for SPARQL query federation.

Consider the FedBench query named `LS2` shown in Listing 1. Imagine for the sake of simplicity that we only had the four FedBench sources presented in Figure 2. Note the data in each dataset is real, the query is taken directly from FedBench and we are thus talking about a real scenario. A TPWSS (e.g., [27; 70; 94; 100]) that retrieves all relevant sources for each individual triple pattern would lead to all sources in the benchmark being queried. This is because the third triple pattern (`?caff ?predicate ?object`) can be answered by all of the datasets. Yet, the complete result set of the query given in Listing 1 can be computed by only querying `DrugBank` and `DBpedia` due to the object-subject join on `?caff` used in the query. This is because the results from the other two data source (i.e., `ChEBI` and Semantic Web Dog Food `SWDF`) will be excluded after performing *joins* between the results of the last two triple patterns of the query.

In the following chapters, we will carry on with this example and show how the approaches we developed, i.e., HiBISCuS partially (by selecting 3 instead of 4 sources) while TBSS completely (by selecting the optimal number of sources, i.e., 2) solved this problem for

```
PREFIX drugbank-drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT ?predicate ?object
WHERE
{
 { drugbank-drug:DB00201 ?predicate ?object . }  //DrugBank
UNION
 {
   drugbank-drug:DB00201 owl:sameAs ?caff .   //DrugBank
   ?caff ?predicate ?object .   //DrugBank, DBpedia, ChEBI, SWDF
}
}
```

Listing 1: Efficient Source Selection Example. FedBench query LS2: Find all properties of Caffeine in Drugbank. Find all entities from all available databases describing Caffeine, return the union of all properties of any of these entities.

the given query. If we had data duplicates (which is not the case in FedBench) then we could combine DAW with TBSS to further prune those sources which contains duplicate data for the given input query [77].

1.3 THE NEED FOR MORE COMPREHENSIVE SPARQL BENCHMARKS

Comprehensive SPARQL benchmarks are mandatory to position new SPARQL query processing systems against existing and help application developers when choosing appropriate systems for a given application. Moreover, benchmark results provide useful insights for system developers and empower them to improve current as well as to develop better systems. However, current SPARQL benchmarks (e.g., [3; 17; 32; 63; 91; 93; 102]) either rely on synthetic data, rely on synthetic queries or are simple in complexity and mostly focus on a limited number of performance criteria.

While synthetic benchmarks allow generating datasets of virtually any size, they often fail to reflect reality [25]. In particular, previous works [25] point out that artificial benchmarks are typically highly structured while real Linked Data sources are less structured. Moreover, synthetic queries most commonly fail to reflect the characteristics of the real queries (i.e., they should show typical requests on the underlying datasets) [8; 68]. Thus, synthetic benchmark results are rarely sufficient to extrapolate the performance of federation engines when faced with real data and real queries. A trend towards benchmarks with real data and real queries (e.g., FedBench [91], DBPSB [63], BioBenchmark [102]) has thus been pursued over the last years but has so far not been able to produce federated SPARQL query benchmarks that reflect the data volumes and query complexity that federated query engines already have to deal with on the Web of Data.

The current benchmarks for SPARQL query execution mostly focused on a single performance criterion, i.e., the query execution time. Thus, they fail to provide results that allow a more fine-grained

```
@PREFIX drugbank-drug:
<http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>.
@PREFIX drugbank-ref:
<http://www4.wiwiss.fu-berlin.de/drugbank/resource/references/>.
@PREFIX dbpedia-resource: <http://dbpedia.org/resource/>.
@PREFIX bio2rdf-pubmed: <http://bio2rdf.org/pubmed>.
drugbank-drug:DB00201 owl:sameAs dbpedia-resource:Caffeine.
drugbank-ref:1002129 owl:sameAs bio2rdf-pubmed:1002129.
```

(a) DrugBank

```
@PREFIX dbpedia-resource: <http://dbpedia.org/resource/>.
dbpedia-resource:Caffeine foaf:name "Caffeine" .
dbpedia-resource:AC_Omonia foaf:name "AC_Omonia" .
```

(b) DBpedia

```
@PREFIX bio2rdf-chebi: <http://bio2rdf.org/chebi:>.
bio2rdf-chebi:21073 chebi:Status bio2rdf-chebi:status-C .
bio2rdf-chebi:21073 rdf:type bio2rdf-chebi:Compound .
```

(c) ChEBI

```
@PREFIX swdf-person: <http://data.semanticweb.org/person/>.
@PREFIX foaf: <http://xmlns.com/foaf/0.1/>.
swdf-person:steve-tjoa foaf:name "Steve Tjoa"
```

(d) Semantic Web Dog Food

Figure 2: Motivating Example. FedBench selected datasets slices. Only relevant prefixes are shown.

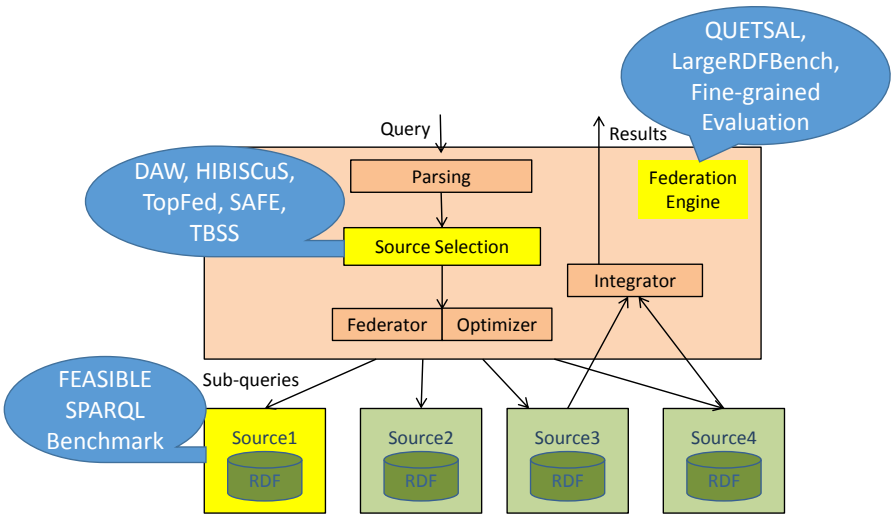


Figure 3: Contributions of this thesis.

evaluation of SPARQL query processing systems to detect the components of systems that need to be improved [62; 87]. For example, performance metrics such as the *completeness and correctness of result sets* and the *effectiveness of source selection* both in terms of *total number of data sources selected*, and the corresponding *source selection time* (which both have a direct impact on the overall query performance) are not addressed in the existing federated SPARQL query benchmarks [62; 87].

1.3.1 *Contributions*

Figure 3 highlights the contributions of this thesis, which addresses problems pertaining to the need for efficient source selection and more comprehensive SPARQL benchmarks as follows:

1. We present HiBISCuS [87], a novel hypergraph based source selection approach which relies on the authority[1] fragment of URIs. HiBISCuS overall relevant sources overestimation is 11.8% (4 times less than FedX [94] and SPLENDID [27]) on FedBench.

2. We present TBSS [88], a novel source selection algorithm based on labelled hypergraphs. TBSS make use of a novel type of data summaries for SPARQL endpoints which relies on most common prefixes for URIs. TBSS is the first source selection approach to remain under 5% overall relevant sources overestimation.

3. We present DAW [77], a duplicate-aware approach for federated query processing over the Web of Data. DAW uses a combination of min-wise independent permutations (MIPs) [20] and

1 URIs Authorities: https://www.ietf.org/rfc/rfc3986.txt

triple selectivity information to estimate the overlap between the results of different sources. DAW has successfully improved the query execution time of the existing federation engines up to 16%.

4. We propose QUETSAL [88], that combines three (HiBISCuS, DAW, and TBSS) source selections approaches with query rewriting to form a complete SPARQL query federation engine. We compare QUETSAL with state-of-the-art federate query engines (FedX [94], SPLENDID [27], ANAPSID [1], and SPLENDID+HiBISCus [87]). Our results show that we have reduced the number of sources selected (without losing recall), the source selection time as well as the overall query runtime by executing many remote joins (i.e., joins shipped to SPARQL endpoints).

5. We present TopFed [90], a specialized TCGA federated query processing engine that exploits the data distribution to perform intelligent source selection while querying over large TCGA SPARQL endpoints. TopFed overall query execution time is half to that of FedX on TCGA queries and endpoints.

6. We present SAFE [52], a novel source selection approach that provides policy-based access to sensitive statistical data represented as distributed RDF Data Cubes. The results shows that SAFE has significantly outperformed FedX in all queries in the context of the presented use-cases.

7. We present LargeRDFBench [79], an open-source benchmark for SPARQL endpoint query federation. To the best of our knowledge, this is the first federated SPARQL query benchmark with real data (from multiple interlinked datasets pertaining to different domains) to encompass more than 1 billion triples. The fine-grained evaluation conducted in LargeRDFBench allows us to pinpoint the restrictions of current SPARQL endpoint federation systems when faced with large datasets, large intermediate results and large result sets. We show that the current ranking of these systems based on simple queries differs significantly from their ranking when on more complex queries.

8. Finally, we present FEASIBLE [78], the first structure- and data-driven feature-based benchmark generation approach from real queries. We show that the performance of triple stores varies greatly across the four basic forms of SPARQL query. Moreover, the features used by FEASIBLE allow for a more fine-grained analysis of the results of benchmarks.

1.4 CHAPTER OVERVIEW

Chapter 2 introduces the basic concepts and notation that are necessary to understand the rest of this thesis. The notation presented in this chapter is used throughout the thesis.

Chapter 3 is based on [84], [78] and discusses the state-of-the-art research work related to this thesis. In particular, Chapter 3 provides a fine-grained evaluation of SPARQL endpoint federation systems. The pros and cons of state-of-the-art SPARQL federation systems are discussed in detail.

Chapter 4 is based on [87] and introduces HiBISCuS, a labelled-hypergraph-based source selection approach which relies on a novel type of data summaries for SPARQL endpoints using the URIs authorities. In the beginning of the chapter, we present the formal framework for modelling SPARQL queries as directed labelled hypergraphs. Afterwards, it explains the data summaries used by the approach followed by the source selection and pruning algorithms. Lastly, the evaluation setup and results are discussed in details.

Chapter 5 is based on [88] and provides the details of TBSS (a trie-based join-aware source selection approach) and QUETSAL (a complete SPARQL query federation engine based on TBSS, HiBiSCuS and DAW). Some of HiBISCuS' limitations are addressed in TBSS by using common name spaces instead of URIs authorities. The TBSS index is discussed next, followed by the source pruning algorithm. Finally, QUETSAL's SPARQL 1.1 query rewriting is explained and the evaluation results are discussed.

Chapter 6 is based on [77] and explains the details of DAW, a duplicate-aware source selection approach for SPARQL query federation over multiple endpoints. Min-wise independent permutations and the DAW index are explained first, followed by DAW's triple pattern-wise source skipping algorithm. Eventually, the results are discussed in details.

Chapter 7 is based on [52] and introduces SAFE, the first (to the best of our knowledge) access policy-based source selection approach for SPARQL endpoint federation. The approach is motivated by providing concrete use cases. The SAFE index and source selection approach is discussed in details. Finally, the evaluation setup and the results are compared with FedX.

Chapter 8 is based on [90] and explains TopFed, a personalized TCGA federation engine which keeps the data locality information in its index. The work is motivated by providing important biological queries, followed by the details of TCGA data conversion and distribution. TopFed's index and the source selection algorithm are explained next and eventually the results are discussed.

Chapter 9 is based on [79] and provides the details of LargeRDFBench, a benchmark for SPARQL endpoint federation. A detail com-

parison of the state-of-the-art federated SPARQL benchmarks is provided. The benchmark datasets and queries are discussed next, followed by the evaluation results.

Chapter 10 is based on [78] and introduces FEASIBLE, a feature-based SPARQL benchmark generation framework. The chapter starts by providing a detailed compassion of the state-of-the-art benchmarks, designed for triple stores evaluations. Next, the benchmark generation framework is explained. Finally, a detailed comparison of the triple stores evaluation is presented and results are discussed.

Finally, Chapter 11 concludes this work and proposes some future work in related areas of research.

2 BASIC CONCEPTS AND NOTATION

This chapter comprises two main parts: (1) we give a brief introduction to the Semantic Web, its core building blocks such as as URIs, RDF(S) and OWL. Furthermore, we will present SPARQL, a formal language to query RDF datasets and the RDF triple stores, (2) we provide formal definitions and notation to the key concepts used throughout this thesis.

2.1 SEMANTIC WEB

The current Web contains a vast amount of information, which in general, could only be understood by humans. Semantic Web is an extension of the current Web that provides an easier way to find, share, reuse and combine information. It empowers machines to not only present but also to process these information.

Tim Berners-Lee outlined this concept in [14] as follows:

> "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

According to the World Wide Web Consortium (W3C):

> "The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners."

In the following, we explain the building blocks of the Semantic Web.

2.1.1 *URIs, RDF*

UNIFORM RESOURCE IDENTIFIER Uniform Resource Identifiers (URIs) build the foundation of the Semantic Web technology. Using URIs we can unambiguously define and reference abstract as well as concrete concepts on a global level. RFC 3986 [1] defines the generic syntax for URIs. This generic URI syntax consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment and can be seen in Figure 4.

1 http://tools.ietf.org/html/rfc3986

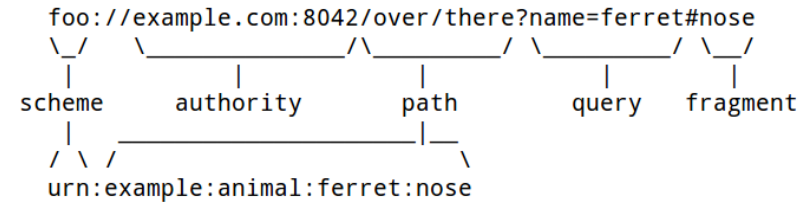


Figure 4: An example URI and its component parts from RFC 3986

The *scheme* and *hierarchy-part* parts are mandatory, whereas the *query* and *fragment* parts are optional. In the context of the Semantic Web, URIs are used to define concepts like *places*, *organizations* or *persons* but also to define relationships between those concepts, e.g. *person-was-born-in-city*, globally and unambiguously. Our contributions, i.e., HiBISCuS [87] (ref. Chapter 4), TBSS [88] (ref. Chapter 5), LargeRDFBench [79] (ref. Chapter 9), and FEASIBLE [78] (ref. Chapter 10) make use of the URIs authorities both for efficient source selection as well as for SPARQL benchmark creation.

RESOURCE DESCRIPTION FRAMEWORK The Resource Description Framework (RDF) is a language to express information or assertions about resources. It is basically a collection of standards [53; 40; 58; 19; 12; 31] released as W3C recommendations. The assertions, which are typically called *triple* in the RDF data model, consist of the following three components:

SUBJECT : The resource over which an assertion is made. Only URIs and blank nodes[2] are allowed to be used as subject of a triple.

PREDICATE : An attribute of a resource or a binary relation which links this resource to another one. Only URIs are valid to be used as predicate of a triple.

OBJECT : Typically, the attribute value or another resource. Valid objects are URIs and blank nodes, but also strings. These strings, also called literals, can be typed or untyped.

Table 1 shows more RDF statements about the resource Muhammad Saleem. The table shows that the resource Muhammad Saleem is the subject of other statements, which give more details about that resource. Also, note that the object of the second and fifth statement (a number and a date) has a trailing datatype. Since URIs can be large, there is a short format for writing them, i.e., by using a prefix. For instance, if we use `http://aksw.org/` as a prefix and give it a label e.g. `aksw`, then resource `http://aksw.org/MuhammadSaleem` can be written as `aksw:MuhammadSaleem`. Similarly, if `http://xmlns.com/foaf/0.1/`

2 A blank node, also call *anonymous node*, is only unambiguous in a local context. It is typically used to group assertions.

Subject	Predicate	Object
aksw:MuhammadSaleem	rdf:type	foaf:Person
aksw:MuhammadSaleem	foaf:age	"31"^^xsd:int
aksw:MuhammadSaleem	foaf:skypeID	"saleem.muhammd"
aksw:MuhammadSaleem	foaf:birthday	"1984-03-03"^^xsd:date
aksw:MuhammadSaleem	foaf:name	"Muhammad Saleem"@en

Table 1: Sample RDF statements.

is used as a prefix with label `foaf`, then the property `http://xmlns.com/foaf/0.1/name` can be written as `foaf:name` in short form. This format is very useful in writing human-readable RDF statements.

2.1.2 *SPARQL Query Language*

"The SPARQL Protocol and RDF Query Language (SPARQL) is a query language and protocol for RDF." [24]. SPARQL is a W3C standard and it is used to query RDF data. A SPARQL query is a combination of triple patterns, their conjunctions (logical "and"), disjunctions (logical "or") and/or a set of optional patterns [101]. A triple pattern is a triple containing subject, predicate, and object parts. The subject can only be a URI, variable, or blank node. The predicate part can only be an IRI[3] (we call it URI for simplicity), or variable, and object can be URI, Blank node, variable, or literal. Note in RDF, a blank node (also called bnode) is a node in an RDF graph representing a resource for which a URI or literal is not given. The resource represented by a blank node is also called an anonymous resource. Examples of the optional patterns are: `FILTER`, `REGEX` and `LANG`. The official syntax of SPARQL 1.0[4] considers operators `OPTIONAL`, `UNION`, `FILTER`, `GRAPH`, `SELECT` and concatenation via a point symbol (`.`), to construct graph pattern expressions (also called Basic Graph Patterns `BGPs`). Operators `SERVICE` and `BINDINGS` are introduced in the SPARQL 1.1[5] federation extension, the former for allowing users to direct a portion of a query to a particular SPARQL endpoint, and the latter for transferring results that are used to constrain a query. The syntax of the language also considers `{ }` to group patterns, and some implicit rules of precedence and association [7]. The results of SPARQL queries can be result sets or RDF graphs. SPARQL has four query forms, specifically `SELECT`, `CONSTRUCT`, `ASK` and `DESCRIBE` [69].

As an example, assume that we want to ask the query "What is the date of birth and Skype id of Muhammad Saleem" to our small knowledge base given in Table 1. Figure 2 shows a SPARQL query to

3 http://www.w3.org/TR/rdf-sparql-query/#sparqlSyntax
4 http://www.w3.org/TR/rdf-sparql-query/
5 http://www.w3.org/TR/sparql11-overview/

```
1 PREFIX aksw: <http://aksw.org/>
2 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3 SELECT ?birthday ?skypeID
4 WHERE
5  {
6    aksw:MuhammadSaleem foaf:birthday ?birthday . //T.P.1
7    aksw:MuhammadSaleem foaf:skypeID ?skypeID .  //T.P.2
8   }
```

Listing 2: SPARQL query to get the the date of birth and Skype id of Muhammad Saleem.

get the required information. This query contains two triple patterns (i.e., T.P.1, T.P.2).

Lines 1 and 2 define prefixes in order to write URIs in their short forms. Line 3 declares the variables that should be rendered to the output of that query, which are two variables `?birthday` and `?skypeID`. SPARQL variables start either with a question mark "`?`", or with a dollar sign "`$`". Line 6 states that for the statement with subject `aksw:MuhammadSaleem` and property `foaf:birthday`, we want the value of its object to be assigned to a variable called `?birthday`. Upon execution, this variable will take the value of `"1984-03-03"^^xsd:date`. The same process is repeated for Line 6 and the results of the projection variables are returned.

2.1.3 *Triplestore*

Triplestores are used to store RDF data. A triplestore is basically a software program capable of storing and indexing RDF data efficiently, in order to enable querying this data easily and effectively. A triplestore for an RDF data is like Relational Database Management System (DBMS) for relational databases.

Most triplestores support SPARQL query language for querying RDF data. Virtuoso[6], Sesame[7], Fuseki [8] and GraphDB[9] are well-known commercial examples of triplestores for desktop and server computers. Since ubiquitous devices usually have less powerful CPU and smaller memory size, there are special version of triplestores that are built to be used on such low-power devices. Androjena[10], RDF On The Go[11], μJena[12] and OpenSesame[13] are examples of such triplestores.

Now we provide the list of basic notation and formal definitions used throughout this thesis.

6 http://virtuoso.openlinksw.com/
7 http://rdf4j.org/sesame/2.8/docs/using+sesame.docbook?view
8 http://jena.apache.org/documentation/serving_data/
9 http://ontotext.com/products/ontotext-graphdb/
10 http://code.google.com/p/androjena/
11 http://code.google.com/p/rdfonthego/
12 http://poseidon.ws.dei.polimi.it/ca/?page_id=59
13 http://bluebill.tidalwave.it/mobile/

2.2 SPARQL SYNTAX, SEMANTIC AND NOTATION

In this section we define the syntax and semantics of SPARQL. Note we are only focusing on the formalizations and notation, necessary to understand the thesis related key concepts and that are used throughout the remaining of this thesis. Note we used some of the definitions from [7].

Definition 1 (RDF Term, RDF Triple and Data Source) *Assume there are pairwise disjoint infinite sets* I, B, *and* L *(IRIs, Blank nodes, and Literals, respectively). Then the RDF term* $RT = I \cup B \cup L$. *The triple* $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ *is called an* RDF triple, *where* s *is called the* subject, p *the* predicate *and* o *the* object. *An* RDF data set *or data source* d *is a set of RDF triples* $d = \{(s_1, p_1, o_1), \ldots, (s_n, p_n, o_n)\}$.

Definition 2 (Query Triple Pattern and Basic Graph Pattern) *By using Definition 1 and assume an infinite set* V *of variables. A tuple* $t \in (I \cup L \cup V \cup B) \times (I \cup V) \times (I \cup L \cup V \cup B)$ *is a triple pattern. A Basic Graph Pattern is a finite set of triple patterns.*

Definition 3 (Basic Graph Pattern syntax) *The syntax of a SPARQL Basic Graph Pattern* BGP *expression is defined recursively as follows:*

1. *A tuple from* $(I \cup L \cup V \cup B) \times (I \cup V) \times (I \cup L \cup V \cup B)$ *is a graph pattern (a triple pattern).*
2. *The expressions (*P_1 *AND* P_2*), (*P_1 *OPTIONAL* P_2*) and (*P_1 *UNION* P_2*) are graph patterns, if* P_1 *and* P_2 *are graph patterns.*
3. *The expression (*P *FILTER* R*) is a graph pattern, if* P *is a graph pattern and* R *is a SPARQL constraint or filter expression.*

Definition 4 (Solution Mapping) *A mapping* μ *from* V *to* RT *is a partial function* $\mu : V \to RT$ *where* $RT = (I \cup B \cup L)$. *For a triple pattern* t, *we denote by* $\mu(t)$ *the pattern obtained by replacing the variables in* t *according to* μ. *The domain of* μ, *denoted by* $\mathrm{dom}(\mu)$, *is the subset of* V *where* μ *is defined. We sometimes write down concrete mappings in square brackets, for instance,* $\mu = [?X \to a, ?Y \to b]$ *is the mapping with* $\mathrm{dom}(\mu) = \{?X, ?Y\}$ *such that,* $\mu(?X) = a$ *and* $\mu(?Y) = b$.

Definition 5 (Triple Pattern Matching) *Let* d *be a data source with set of RDF terms* RT, *and* t *a triple pattern of a SPARQL query. The evaluation of* t *over* d, *denoted by* $[\![t]\!]d$ *is defined as the set of mappings* $[\![t]\!]d = \{\mu : V \to RT \mid \mathrm{dom}(\mu) = \mathrm{var}(P) \text{ and } \mu(P) \subseteq d\}$. *If* $\mu \in [\![t]\!]d$, *we say that* μ *is a solution for* t *in* d. *If a data source* d *has at least one solution for a triple pattern* t, *then we say* d *matches* t.

Definition 6 (Relevant Source and Set) *A data source* $d \in D$ *is* relevant *(also called* capable*) for a triple pattern* $t_i \in T$ *if at least one triple contained in* d *matches* t_i. *The relevant source set* $R_i \subseteq D$ *for* t_i *is the set that contains all sources that are relevant for that particular triple pattern.*

Definition 7 (Triple Pattern-wise Source Selection) *The goal of the Triple Pattern-wise Source Selection (TPWSS) is to identify the set of relevant sources against individual triple patterns of a query.*

Definition 8 (Total Triple Pattern-wise Sources Selected) *Let* $q = \{t_1, \ldots, t_m\}$ *be a SPARQL query containing triple patterns* $t_1, \ldots, t_m$, $\mathcal{R} = \{R_{t_1}, \ldots, R_{t_m}\}$ *be the corresponding relevance set containing relevant data sources sets* $R_{t_1}, \ldots, R_{t_m}$ *for triple patterns* $t_1, \ldots, t_m$, *respectively. We define TTP-WSS* $= \forall_{R_{t_i} \in \mathcal{R}} \sum |R_{t_i}|$ *be the total triple pattern-wise sources selected for query q, i.e., the sum of the magnitudes of relevant data sources sets over all individual triple patterns q.*

The total triple pattern-wise sources selected for the query given in Listing 1 is six, i.e., a single source for first two triple patterns and four sources for the last triple pattern, summing up to a total of six sources.

Definition 9 (Number of Sources Span) *The number of sources that potentially contribute to the query result set (sources span for short) are those that are* relevant *to at least one triple pattern in the query. However, since triple patterns with common query predicates such as* `rdf:type` *and* `owl:sameAs` *are likely to be found in all data sources (i.e., all sources are* relevant*), we only count a source if it is also* relevant *to at least one more triple pattern in the same query.*

SPARQL Query as Directed Hypergraph

We represent each basic graph pattern (BGP) of a SPARQL query as a directed hypergraph (DH). Note this is one of the contribution of HiBISCuS [87] presented in Chapter 4. We chose this representation because it allows representing property-property joins, which previous works [3; 30] do not allow to model. The DH representation of a BGP is formally defined as follows:

Definition 10 *Each basic graph patterns* BGP_i *of a SPARQL query can be represented as a DH* $HG_i = (V, E, \lambda_{vt})$, *where*

- $V = V_s \cup V_p \cup V_o$ *is the set of vertices of* HG_i, V_s *is the set of all subjects in* HG_i, V_p *the set of all predicates in* HG_i *and* V_o *the set of all objects in* HG_i*;*
- $E = \{e_1, \ldots, e_t\} \subseteq V^3$ *is a set of directed hyperedges (short: edge). Each edge* $e = (v_s, v_p, v_o)$ *emanates from the triple pattern* $<v_s, v_p, v_o>$ *in* BGP_i. *We represent these edges by connecting the head vertex* v_s *with the tail hypervertex* (v_p, v_o). *We use* $E_{in}(v) \subseteq E$ *and* $E_{out}(v) \subseteq E$ *to denote the set of incoming and outgoing edges of a vertex* v*;*
- λ_{vt} *is a vertex-type-assignment function. Given a vertex* $v \in V$, *its vertex type can be 'star', 'path', 'hybrid', or 'sink' if this vertex participates in at least one join. A 'star' vertex has more than one outgoing*

```
SELECT DISTINCT * WHERE
{
?drug :description ?drugDesc.
?drug :drugType :smallMolecule.
?drug :keggCompoundId ?compound.
?enzyme :xSubstrate ?compound.
?Chemicalreaction :xEnzyme ?enzyme.
?Chemicalreaction :equation ?ChemicalEquation.
?Chemicalreaction :title ?ReactionTitle .
}
```

Listing 3: Example SPARQL query. Prefixes are ignored for simplicity

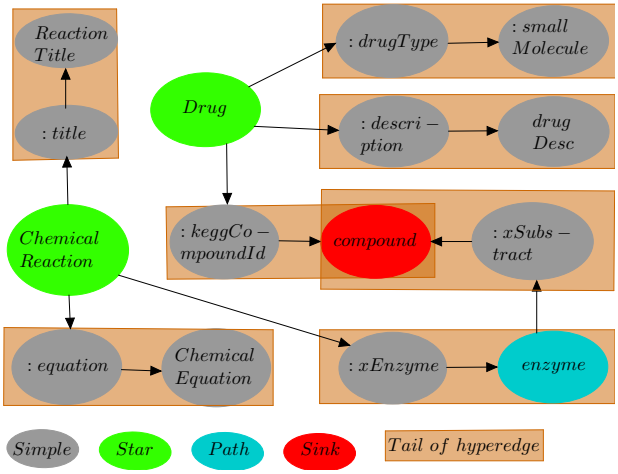


Figure 5: DH representation of the SPARQL query given in Listing

edge and no incoming edge. A 'path' vertex has exactly one incoming and one outgoing edge. A 'hybrid' vertex has either more than one incoming and at least one outgoing edge or more than one outgoing and at least one incoming edge. A 'sink' vertex has more than one incoming edge and no outgoing edge. A vertex that does not participate in any join is of type 'simple'.

The representation of a complete SPARQL query as a DH is the union of the representations of query's BGPs. As an example, the DH representation of the query in Listing 3 is shown in Figure 5. Based on the DH representation of SPARQL queries we can define the following features of SPARQL queries:

Definition 11 (Number of Triple Patterns) *From Definition 10, the total number of triple patterns in a* BGP_i *is equal to the number of hyperedges* $|E|$ *in the DH representation of the* BGP_i.

Definition 12 (Number of Join Vertices) *Let* $ST = \{st_1, \ldots, st_j\}$ *be the set of vertices of type 'star',* $PT = \{pt_1, \ldots, pt_k\}$ *be the set of vertices of type 'path',* $HB = \{hb_1, \ldots, hb_l\}$ *be the set of vertices of type 'hybrid', and* $SN = \{sn_1, \ldots, sn_m\}$ *be the set of vertices of type 'sink' in a DH representation of a SPARQL query, then the total number of join vertices in the query* $\#JV = |ST| + |PT| + |HB| + |SN|$.

Definition 13 (Join Vertex Degree) *Based on the DH representation of SPARQL queries, the join vertex degree of a vertex v is* $JVD(v) = |E_{in}(v)| + |E_{out}(v)|$*, where* $E_{in}(v)$ *resp* $E_{out}(v)$ *is the set of incoming resp. outgoing edges of* v*.*

Definition 14 (Triple Pattern Selectivity) *Let* tp_i *be a triple pattern and* d *be a* relevant *source for* tp_i*. Furthermore, let* N *be the total number of triples in* d *and* N_m *be the total number of triples in* d *that matches* tp_i*, then the selectivity of* tp_i *w.r.t.* d *is* $Sel(tp_i, d) = N_m/N$*.*

3

STATE OF THE ART

This chapter is based on [84], [78] and discusses the state-of-the-art research work related to this thesis. In particular, this chapter provides a fine-grained evaluation of SPARQL endpoint federation systems. Furthermore, the chapter presents the results of a public survey which provides a crisp overview of categories of SPARQL federation systems as well as their implementation details, features, and supported SPARQL clauses. The pros and cons of state-of-the-art SPARQL federation systems are discussed in details. We address some of the key shortcomings (in particular those related to source selection and benchmarking) of the existing approaches in this thesis and present the corresponding results in the subsequent chapters.

Given the importance of federated query processing over the Web of Data, it is critical to provide fine-grained evaluations to assess the quality of state-of-the-art implementations of federated SPARQL query engines by comparing them against common criteria through an open benchmark. Such fine-grained evaluation results are valuable when positioning new federation systems against existing. In addition, these results help application developers when choosing appropriate systems for a given application as they allow them to select federation systems through a comparison of their performance against their criteria of interest. Moreover, such fine-grained results provide useful insights for system developers and empower them to improve current federation systems as well as to develop better systems.

Current evaluations [1; 27; 61; 73; 89; 94; 100] of SPARQL query federation systems compare some of the federation systems based on the central criterion of overall query runtime. While optimizing the query runtime of federation systems is the ultimate goal of the research performed on this topic, the granularity of current evaluations fails to provide results that allow understanding why the query runtimes of systems can differ drastically and are thus insufficient to detect the components of systems that need to be improved. For example, key metrics such as a smart source selection in terms of the *total number of data sources selected*, the *total number of SPARQL ASK requests used*, and the *source selection time* have a direct impact on the overall query performance but are not addressed in the existing evaluations. For example, the over-estimation of the total number of relevant data sources will generate extra network traffic, may result in increased query execution time. The SPARQL ASK queries returns boolean value indicating whether a query pattern matches or not. The greater the number

of SPARQL ASK requests used for source selection, the higher the source selection time and therefore overall query execution time. Furthermore, as pointed out by [62], the current testbeds [17; 63; 92; 93] for evaluating, comparing, and eventually improving SPARQL query federation systems still have some limitations. Especially, the *partitioning of data* as well as *the SPARQL clauses* used cannot be tailored sufficiently, although they are known to have a direct impact on the behaviour of SPARQL query federation systems.

The aim of this chapter is to experimentally evaluate a large number of SPARQL 1.0 query federation systems within a more fine-granular setting in which we can measure the time required to complete different steps of the SPARQL query federation process. To achieve this goal, we conducted a public survey[1] and collected information regarding 14 existing federated system implementations, their key features, and supported SPARQL clauses. Eight of the systems which participated in this survey are publicly available. However, two out of the eight with public implementation do not make use of the SPARQL endpoints and were thus not considered further in this study. Note this thesis is based on query federation over multiple SPARQL endpoints or SPARQL endpoint federation for short.

In the next step and like in previous evaluations, we compared the remaining six systems [1; 27; 57; 70; 94; 100] with respect to the traditional performance criterion that is the query execution time using the commonly used benchmark FedBench. In addition, we also compared these six systems with respect to their *answer completeness*, source selection approach in terms of the *total number of sources* they selected, the *total number of SPARQL ASK requests* they used and *source selection time*. For the sake of completeness, we also performed a comparative analysis (based on the survey outcome) of the key functionality of the 14 systems which participated in our survey. Furthermore, we also discussed the systems that did not participate or published after the survey. The most important outcomes of this survey are presented in Section 3.3.[2]

To provide a quantitative analysis of the effect of *data partitioning* on the systems at hand, we extended both FedBench [92] and SP^{2}Bench [93] by distributing the data upon which they rely. To this end, we used the slice generation tool[3] described in [77]. This tool allows creating any number of subsets of a given dataset (called *slices*) while controlling the number of slices, the amount of overlap between the slices as well as the size distribution of these slices. The resulting slices were distributed across various data sources (SPARQL endpoints) to simulate a highly federated environment. In our experiments, we made use of both FedBench [92] and SP^{2}Bench [93]

1 Survey: http://goo.gl/iXvKVT, Results: http://goo.gl/CNW5UC
2 All survey responses can be found at http://goo.gl/CNW5UC.
3 https://code.google.com/p/fed-eval/wiki/SliceGenerator

queries to ensure that we *cover the majority of the SPARQL query types and clauses*.

Our main contributions of this chapter are summarized as follows:

- We present the results of a public survey which allows us to provide a crisp overview of categories of SPARQL federation systems as well as provide their implementation details, features, and supported SPARQL clauses.
- We present (to the best of our knowledge) the most comprehensive experimental evaluation of open-source SPARQL federations systems in terms of their source selection and overall query runtime using in two different evaluation setups.
- Along with the central evaluation criterion (i.e., the overall query runtime), we measure three further criteria, i.e., the total number of sources selected, the total number of SPARQL ASK requests used, and the source selection time. By these means, we obtain deeper insights into the behaviour of SPARQL federation systems.
- We extend both FedBench and SP^{2}Bench to mirror highly distributed data environments and test SPARQL endpoint federation systems for their parallel processing capabilities.
- We provide a detailed discussion of experimental results and reveal novel insights for improving existing and future federation systems.
- Our survey results points to research opportunities in the area of federated semantic data processing.

The rest of this chapter is structured as follows: In Section 3.1, we give an overview of existing federated SPARQL query system evaluations. Here, we focus on the description of different surveys/evaluations of SPARQL query federation systems and argue for the need of a new fine-grained evaluation of federated SPARQL query engines. Section 3.2 gives an overview of benchmarks for SPARQL query processing engines. In addition, we provide reasons for our benchmark selection in this evaluation. Section 3.3 provides a detailed description of the design of and the responses to our public survey of the SPARQL query federation. Section 3.4 provides an introduction to selected approaches for experimental evaluation. Section 3.6 describes our evaluation framework and experimental results, including key performance metrics, a description of the used benchmarks (FedBench, SP^{2}Bench, SlicedBench), and the data slice generator. Finally, Section 3.7 provides our further discussion of the results.

3.1 FEDERATION SYSTEMS EVALUATIONS

Several SPARQL query federation surveys have been developed over the last years. Rakhmawati et al. [73] present a survey of SPARQL endpoint federation systems in which the details of the query federation process along with a comparison of the query evaluation strategies used in these systems. Moreover, systems that support both SPARQL 1.0 and SPARQL 1.1 are explained. However, this survey do not provide any experimental evaluation of the discussed SPARQL query federation systems. In addition, the system implementation details resp. supported features are not discussed in much detail. We address these drawbacks in Tables 3 resp. 4. Hartig et al. [36] provide a general overview of Linked Data federation. In particular, they introduce the specific challenges that need to be addressed and focus on possible strategies for executing Linked Data queries. However, this survey do not provide an experimental evaluation of the discussed SPARQL query federation systems. Umbrich et al. [45] provide a detailed study of the recall and effectiveness of Link Traversal Querying for the Web of Data. Schwarte et al. [96] present an experimental study of large-scale RDF federations on top of the Bio2RDF data sources using a particular federation system, i.e., FedX [94]. They focus on design decisions, technical aspects, and experiences made in setting up and optimizing the Bio2RDF federation. Betz et al. [15] identify various drawbacks of federated Linked Data query processing. The authors propose that Linked Data as a service has the potential to solve some of the identified problems. Görlitz et al. [33] present limitations in Linked Data federated query processing and implications of these limitations. Moreover, this chapter presents a query optimization approach based on semi-joins and dynamic programming. Ladwig et al. [54] identify various strategies while processing federated queries over Linked Data. Umbrich et al. [99] provide an experimental evaluation of the different data summaries used in live query processing over Linked Data. Montoya et al. [62] provide a detail discussion of the limitations of the existing testbeds used for the evaluation of SPARQL query federation systems. Some other experimental evaluations [1; 27; 61; 62; 89; 94; 100] of SPARQL query federation systems compare some of the federation systems based on their overall query runtime. For example, Gorlitz et al. [27] compare their approach with three other approaches ([70; 94], AliBaba[4], and RDF-3X 0.3.4.22[7]. An extension of ANAPSID presented in [61] compares ANAPSID with FedX using 10 FedBench-additional complex

4 SesameAliBaba:http://www.openrdf.org/alibaba.jsp) using a subset of the queries from FedBench. Furthermore, they measure the effect of the information in VoiD[5] descriptions on the accuracy of their source selection. Acosta et al. [1] compare their approach performance with Virtuoso SPARQL endpoints, ARQ 2.8.8. BSD-style21[6]

7 http://www.mpi-inf.mpg.de/neumann/rdf3x/

queries. Schwarte et al. [94] compare FedX performance with AliBaba and DARQ using a subset of the FedBench queries. Wang et al. [100] evaluate the performance of LHD with FedX and SPLENDID using the Berlin SPARQL Benchmark (BSBM) [17].

All experimental evaluations above compare only a small number of SPARQL query federation systems using a subset of the queries available in current benchmarks with respect to a single performance criterion (query execution time). Consequently, they fail to provide deeper insights into the behaviour of these systems in different steps (e.g., source selection) required during the query federation. In this work, we evaluate six open-source federated SPARQL query engines experimentally on two different evaluation frameworks. To the best of our knowledge, this is currently the most comprehensive evaluation of open-source SPARQL query federation systems. Furthermore, along with central performance criterion of query runtime, we compare these systems with respect to their source selection. Our results show (section 3.6) that the different steps of the source selection affect the overall query runtime considerably. Thus, the insights gained through our evaluation w.r.t. to these criteria provide valuable findings for optimizing SPARQL query federation.

3.2 BENCHMARKS

Different benchmarks have been proposed to compare triple stores and federated systems for their query execution capabilities and performance. Table 2 provides a detailed summary of the characteristics of the most commonly used benchmarks as well as of two real query logs. All benchmark executions and result set computations were carried out on a machine with 16 GB RAM and a 6-Core i7 3.40 GHz CPU running Ubuntu 14.04.2. All synthetic benchmarks were configured to generate 10 million triples. We ran LUBM [32] on OWLIM-Lite as it requires reasoning. All other benchmarks were ran on virtuoso 7.2 with NumberOfBuffers = 1360000, and MaxDirtyBuffers = 1000000.

In the following, we compare state-of-the-art SPARQL benchmarks w.r.t. the features shown in Table 2.

LUBM was designed to test the triple stores and reasoners for their reasoning capabilities. It is based on a customizable and deterministic generator for synthetic data. The queries included in this benchmark commonly lead to query results sizes ranges from 2 to 3200, query triple patterns ranges from 1 to 6, and all the queries consist of a single BGP. LUBM includes a fixed number of `SELECT` queries (i.e., 15) where none of the clauses shown in Table 2 is used.

The Berlin SPARQL Benchmark (BSBM) [17] uses a total of 125 query templates to generate any number of SPARQL queries for benchmarking. Multiple use cases such as explore, update, and business intelligence are included in this benchmark. Furthermore, it also in-

Table 2: Comparison of SPARQL benchmarks **F-DBP** = FEASIBLE Benchmarks from DBpedia query log, **F-SWDF** = FEASIBLE Benchmark from Semantic Web Dog Food query log, **LRB** = LargeRDFBench, **TPs** = Triple Patterns, **JV** = Join Vertices, **MJVD** = Mean Join Vertices Degree, **MTPS** = Mean Triple Patterns Selectivity, **S.D.** = Standard Deviation, **U.D.** = Undefined due to queries timeout of 1 hour). Runtime(ms) and *SPARQL federation benchmarks.

		LUBM	BSBM	SP2Bench	WatDiv	DBPSB	F-DBP	F-SWDF	FedBench*	LRB*
	#Queries	15	125	12	125	125	125	125	25	32
Forms (%)	SELECT	100	80	91.67	100	100	95.2	92.8	100	100
	ASK	0	0	8.33	0	0	0	2.4	0	0
	CONSTRUCT	0	4	0	0	0	4	3.2	0	0
	DESCRIBE	0	16	0	0	0	0.8	1.6	0	0
Clauses (%)	UNION	0	8	16.67	0	36	40.8	32.8	12	18.75
	DISTINCT	0	24	41.6	0	100	52.8	50.4	0	28.21
	ORDER BY	0	36	16.6	0	0	28.8	25.6	0	9.37
	REGEX	0	0	0	0	4	14.4	16	0	3.12
	LIMIT	0	36	8.33	0	0	38.4	45.6	0	12.5
	OFFSET	0	4	8.33	0	0	18.4	20.8	0	0
	OPTIONAL	0	52	25	0	32	30.4	32	4	25
	FILTER	0	52	58.3	0	48	58.4	29.6	4	31.25
	GROUP BY	0	0	0	0	0	0.8	19.2	0	0
Results	Min	3	0	1	0	197	1	1	1	1
	Max	1.3E+4	31	4.3E+7	4.1E+9	4.6E+6	1.4E+6	3.0E+5	9054	3.E+5
	Mean	4.9E+3	8.3	4.5E+6	3.4E+7	3.2E+5	5.2E+4	9091	529	5.9E+4
	S.D.	1.1E+4	9.03	1.3E+7	3.7E+8	9.5E+5	1.9E+5	4.7E+4	1764	1.1E+5
BGPs	Min	1	1	1	1	1	1	0	1	1
	Max	1	5	3	1	9	14	14	2	2
	Mean	1	2.8	1.5	1	2.69	3.17	2.68	1.16	1.43
	S.D.	0	1.70	0.67	0	2.43	3.55	2.81	0.37	0.5
TPs	Min	1	1	1	1	1	1	0	2	2
	Max	6	15	13	12	12	18	14	7	12
	Mean	3	9.32	5.9	5.3	4.5	4.8	3.2	4	6.6
	S.D.	1.81	5.17	3.82	2.60	2.79	4.39	2.76	1.25	2.6
JV	Min	0	0	0	0	0	0	0	0	0
	Max	4	6	10	5	3	11	3	5	6
	Mean	1.6	2.88	4.25	1.77	1.21	1.29	0.52	2.52	3.43
	S.D.	1.40	1.80	3.79	0.99	1.12	2.39	0.65	1.26	1.36
MJVD	Min	0	0	0	0	0	0	0	0	0
	Max	5	4.5	9	7	5	11	4	3	6
	Mean	2.02	3.05	2.41	3.62	1.82	1.44	0.96	2.14	2.56
	S.D.	1.29	1.63	2.26	1.40	1.43	2.13	1.09	0.56	0.76
MTPS	Min	3.2E-4	9.4E-8	6.5E-5	0	1.1E-5	2.8E-9	1.0E-5	0.001	0.001
	Max	0.432	0.045	0.53	0.011	1	1	1	1	1
	Mean	0.01	0.01	0.22	0.004	0.119	0.140	0.291	0.05	0.10
	S.D.	0.074	0.01	0.20	0.002	0.22	0.31	0.32	0.092	0.14
Runtime	Min	2	5	7	3	11	2	4	50	159
	Max	3200	99	7.1E+5	8.8E+8	5.4E+4	3.2E+4	4.1E+4	1.2E+4	>1hr
	Mean	437	9.1	2.8E+5	4.4E+8	1.0E+4	2242	1308	1987	U.D.
	S.D.	320	14.5	5.2E+5	2.7E+7	1.7E+4	6961	5335	3950	U.D.

cludes many of the important SPARQL clauses of Table 2. However, the queries included in this benchmark are rather simple with an average query runtime of 9.1 ms and largest query result set size equal to 31.

SP²Bench mirrors vital characteristics (such as power law distributions or Gaussian curves) of the data in the DBLP bibliographic database. The queries given in benchmark are mostly complex. For example, the mean (across all queries) query result size is above one million and the query runtimes are very large (see Table 2).

The Waterloo SPARQL Diversity Test Suite (WatDiv) [3] addresses the limitations of previous benchmarks by providing a synthetic data and query generator to generate large number of queries from a total of 125 queries templates. The queries cover both simple and complex categories with varying number of features such as result set sizes, total number of query triple patterns, join vertices and mean join vertices degree. However, this benchmark is restricted to conjunctive `SELECT` queries (single BGPs). This means that non-conjunctive SPARQL queries (e.g., queries which make use of the `UNION` and `OPTIONAL` features) are not considered. Furthermore, WatDiv does not consider other important SPARQL clauses, e.g., `FILTER` and `REGEX`. However, our analysis of the query logs of DBpedia3.5.1 and SWDF given in table 2 shows that 20.1% resp. 7.97% of the DBpedia queries make use of `OPTIONAL` resp. `UNION` clauses. Similarly, 29.5% resp. 29.3% of the SWDF queries contain `OPTIONAL` resp. `UNION` clauses.

While the distribution of query features in the benchmarks presented so far is mostly static, the use of different SPARQL clauses and triple pattern join types varies greatly from data set to data set, thus making it very difficult for any synthetic query generator to reflect real queries. For example, the DBpedia and SWDF query log differ significantly in their use of `DESCRIBE` (41.1% for SWDF vs 0.02% for DBpedia), `FILTER` (0.72% for SWDF vs 93.3% for DBpedia) and `UNION` (29.3% for SWDF vs 7.97% for DBpedia) clauses. Similar variations have been reported in [8] as well. To address this issue, the DBpedia SPARQL Benchmark (DBPSB) [63] (which generates benchmark queries from query logs) was proposed. However, is benchmark does not consider key query features (i.e., number of join vertices, mean join vertices degree, mean triple pattern selectivities, the query result size and overall query runtimes) while selecting query templates. Note that previous works [3; 30] pointed that these query features greatly affect the triple stores performance and thus should be considered while designing SPARQL benchmarks.

In this thesis we present FEASIBLE, a benchmark generation framework which is able to generate a customizable benchmark from any set of queries, esp. from query logs. FEASIBLE addresses the drawbacks on previous benchmark generation approaches by taking all of the important SPARQL query features of Table 2 into consideration

when generating benchmarks. In Chapter 10, we present FEASIBLE in detail.

The aforementioned benchmarks have focused on the problem of query evaluation over local, centralised repositories. Hence, these benchmarks do not consider federated queries over multiple interlinked datasets hosted by different SPARQL endpoints. FedBench [92] is designed explicitly to evaluate SPARQL query federation tasks on real-world datasets with queries resembling typical requests on these datasets. Furthermore, this benchmark also includes a dataset and queries from SP^{2}Bench. FedBench were the only (to the best of our knowledge) federation benchmark that encompasses real-world datasets, commonly used queries and distributed data environment. Furthermore, it is commonly used in the evaluation of SPARQL query federation systems [27; 61; 77; 94]. However, the real queries (excluding synthetic SP^{2}Bench benchmark queries) are low in complexity (Table 2). The number of Triple Patterns included in the query ranges from 2 to 7. Consequently, the standard deviations of the number of Join Vertices (JV) and the Mean Join Vertices Degrees (MJVD) are on the lower side. The query result set sizes are small (maximum 9054, with average of 529 results). The query triple patterns are not highly selective in general. The important SPARQL clauses such `DISTINCT`, `ORDER BY` and `REGEX` are not used (ref. Table 2). Moreover, the SPARQL `OPTIONAL` and `FILTER` clauses are only used in a single query (i.e., LS7 of FedBench). Most importantly, the average query execution is small (about 2 seconds on average ref. Section 3.6.3.5). Finally, FedBench rely only on the number of endpoints requests and the query execution time as performance criteria. These limitations make it difficult to extrapolate how SPARQL query federation engines will perform when faced with the growing amount of data available on the Data Web based on FedBench results. Furthermore, a more fine-grained evaluation of the federation engines, to detect the components that need to be improved is not possible [62].

SPLODGE [30] is a heuristic for automatic generation of federated SPARQL queries which is limited to conjunctive BGPs. Non-conjunctive queries that make use of the SPARQL `UNION`, `OPTIONAL` clauses are not considered. Thus, the generated set of synthetic queries fails to reflect the characteristics of the real queries. For example, the DBpedia query log [68] shows that 20.87%, 30.02% of the real queries contains SPARQL `UNION` and `FILTER` clauses, respectively. However, both of these clauses are not considered in SPLODGE queries generation. Moreover, the use of different SPARQL clauses and triple pattern join types greatly varies from one dataset to another dataset, thus making it almost impossible for automatic query generator to reflect the reality. For example, the DBpedia and Semantic Web Dog Food (SWDF) query log [8] shows that the use of the SPARQL `LIMIT` (27.99% for SWDF vs 1.04% for DBpedia) and `OPTIONAL` (0.41% for

SWDF vs 16.61% for DBpedia) clauses greatly varies for these two datasets.

To address the limitations of federated SPARQL benchmarks, we propose LargeRDFBench, a billion-triple benchmark which encompasses a total of 13 real, interconnected datasets and real queries of varying complexities (see Table 2). Our benchmark includes all of the 14 SPARQL endpoint federation queries (which we named *simple queries*) from FedBench, as they are useful but not sufficient all alone. In addition, we provide 10 complex and 8 Big Data queries, which lead to larger result sets and intermediary results. Beside the central performance criterion, i.e., the query execution time, our benchmark includes result set completeness and correctness, effective source selection in terms of the total number of data sources selected, the total number of SPARQL ASK requests used and the corresponding source selection time. Our evaluation results (ref. Chapter 9) suggest that the performance of current SPARQL query federation systems on simple queries (i.e., FedBench queries) does not reflect the systems' performance on more complex queries. In addition, none of the state-of-the-art SPARQL query federation is able to fully answer the real use-case Big Data queries. In Chapter 9, we will discuss LargeRDFBench in details.

3.3 FEDERATED ENGINES PUBLIC SURVEY

In order to provide a comprehensive overview of existing SPARQL federation engines, we designed and conducted a survey of SPARQL query federation engines. In this section, we present the principles and ideas behind the design of the survey as well as its results and their analysis.

3.3.1 *Survey Design*

The aim of the survey was to compare the existing SPARQL query federation engines, regardless of their implementation or code availability. To reach this aim, we interviewed domain experts and designed a survey with three sections: system information, requirements, and supported SPARQL clauses.[8]

The system information section of the survey includes implementation details of the SPARQL federation engine such as:

- **URL of the paper, engine implementation:** Provides the URL of the related scientific publication or URL to the engine implementation binaries/code.
- **Code availability:** Indicates the disclosure of the code to the public.

8 The survey can be found at http://goo.gl/iXvKVT.

- **Implementation and licensing:** Defines the programming language and distribution license of the framework.
- **Type of source selection:** Defines the source selection strategy used by the underlying federation system.
- **Type of join(s) used for data integration:** Shows the type of *joins* used to integrate sub-queries results coming from different data sources.
- **Use of cache:** Shows the usage of cache for performance improvement.
- **Support for catalog/index update:** Indicates the support for automatic index/catalog update.

The questions from the *requirements* section assess SPARQL query federation engines for the key features/requirements that a developer would require from such engines. These include:

- **Result completeness:** Assuming that the SPARQL endpoints return complete results for any given SPARQL1.0 sub-query that they have to process. Does your implementation then guarantee that your engine will return complete results for the input query (100% recall) or is it possible that it misses some of the solutions (for example due to the source selection, join implementation, or using an out-of-date index)?. Please note that a 100% recall cannot be assured with an index that is out of date.
- **Policy-based query planning:** Most federation approaches target open data and do not provide restrictions (according to different user access rights) on data access during query planning. As a result, a federation engine may select a data source for which the requester is not authorized, thus overestimating the data sources and increasing the overall query runtime. Does your system have the capability of taking into account the privacy information (e.g., different graph-level access rights for different users, etc.) during query planning?
- **Support for partial results retrieval:** In some cases the query results can be too large and result completeness (i.e., 100% recall) may not be desired, rather partial but fast and/or quality query results are acceptable. Does the federation engine provide such functionality where a user can specify a desired recall (less than 100%) as a threshold for fast result retrieval? It is worth noticing that this is different from limiting the results using SPARQL LIMIT clause as it restricts the number of results to some fixed value while in partial result retrieval the number of retrieved results are relative to the actual total number of results.

- **Support for no-blocking operator/adaptive query processing:** SPARQL endpoints are sometimes blocked or down or exhibit high latency. Does the federation engine support non-blocking joins (where results are returned based on the order in which the data arrives, not in the order in which data being requested) and able to change its behavior at runtime by learning behavior of the data providers?

- **Support for provenance information:** Usually, SPARQL query federation systems integrate results from multiple SPARQL endpoints without any provenance information, such as how many results were contributed by a given SPARQL endpoint or which of the results are contributed by each of the endpoint. Does the federation engine provide such provenance information?

- **Query runtime estimation:** In some cases a query may have a longer runtime (e.g., in the order of minutes). Does the federation engine provide means to approximate and display (to the user) the overall runtime of the query execution in advance?

- **Duplicate detection:** Due to the decentralized architecture of Linked Data Cloud, a sub-query might retrieve results that were already retrieved by another sub-query. For some applications, the former sub-query can be skipped from submission (federation) as it will only produce overlapping triples. Does the federation engine provide such a duplicate-aware SPARQL query federation? Note that this is the duplicate detection before sub-query submission to the SPARQL endpoints and the aim is to minimize the number of sub-queries submitted by the federation engine.

- **Top-K query processing:** Is the federation engine able to rank results based on the user's preferences (e.g., his/her profile, his/her location, etc.)?

The *supported SPARQL clauses* section assess existing SPARQL query federation engines w.r.t. the list of supported SPARQL clauses. The list of the SPARQL clauses is mostly based on the characteristics of the BSBM benchmark queries [17]. The summary of the used SPARQL clauses can be found in Table 5.

The survey was open and free for all to participate in. To contact potential participants, we used Google Scholar to retrieve papers that contained the keywords "SPARQL" and "query federation". After a manual filtering of the results, we contacted the main authors of the papers and informed them of the existence of the survey while asking them to participate. Moreover, we sent messages to the W3C Linked

Open Data mailing list[9] and Semantic Web mailing list[10] with a request to participate. The survey was opened for two weeks.

3.3.2 *Discussion of the survey results*

Based on our survey results[11], existing SPARQL query federation approaches can be divided into three main categories (see Table 3).

1. query federation over multiple sparql endpoints: In this type of approaches, RDF data is made available via SPARQL endpoints. The federation engine makes use of endpoint URLs to federate sub-queries and collect results back for integration. The advantage of this category of approaches is that queries are answered based on original, up-to-date data with no synchronization of the copied data required [36]. Moreover, the execution of queries can be carried out efficiently because the approach relies on SPARQL endpoints. However, such approaches are unable to deal with the data provided by the whole of LOD Cloud because sometimes Linked Data is not exposed through SPARQL endpoints.

2. query federation over linked data: This type of approaches relies on the Linked Data principles[12] for query execution. The set of data sources which can contribute results into the final query result-set is determined by using URI lookups during the query execution itself.

9 public-lod@w3.org
10 semantic-web@w3.org
11 Available at http://goo.gl/CNW5UC
12 http://www.w3.org/DesignIssues/LinkedData.html

Table 3: Overview of implementation details of federated SPARQL query engines (**SEF** = SPARQL Endpoints Federation, **DHTF** = DHT Federation, **LDF** = Linked Data Federation, **C.A.** = Code Availability, A.G.P.L. Affero General Public License, L.G.P.L. = Lesser General Public License, **S.S.T.** = Source Selection Type, **I.U.** = Index/catalog Update, (A+I) = SPARQL ASK and Index/catalog, (C+L) = Catalog and online discovery via Link-traversal), VENL = Vectored Evaluation in Nested Loop, **AGJ** = Adaptive Group Join, **ADJ** = Adaptive Dependent Join, **RMHJ** = Rank-aware Modification of Hash Join, NA = Not Applicable

Systems	Category	C.A	Implementation	Licencing	S.S.T	Join Type	Cache	I.U
FedX [94]	SEF	✓	Java	GNU A.G.P.L	index-free	bind (VENL)	✓	NA
LHD [100]	SEF	✓	Java	MIT	hybrid (A+I)	hash/ bind	✗	✗
SPLENDID [27]	SEF	✓	Java	L.G.P.L	hybrid (A+I)	hash/ bind	✗	✗
FedSearch [5]	SEF	✗	Java	GNU A.G.P.L	hybrid (A+I)	bind (VENL), pull based rank join (RMHJ)	✓	NA
GRANATUM [38]	SEF	✗	Java	yet to decide	index only	nested loop	✗	✗
Avalanche [11]	SEF	✗	Python, C, C++	yet to decide	index only	distributed, merge	✓	✗
DAW [77]	SEF	✗	Java	GNU G.P.L	hybrid (A+I)	based on underlying system	✓	✗
ANAPSID [1]	SEF	✓	Python	GNU G.P.L	hybrid (A+I)	AGJ, ADJ	✗	✓
ADERIS [57]	SEF	✓	Java	Apache	Index only	index-based nested loop	✗	✗
DARQ [70]	SEF	✓	Java	GPL	Index only	nested loop, bound	✗	✗
LDQPS [54]	LDF	✗	Java	Scala	hybrid (C+L)	symmetric hash	✗	✗
SIHJoin [55]	LDF	✗	Java	Scala	hybrid (C+L)	symmetric hash	✗	✗
WoDQA [2]	LDF	✓	Java	GPL	hybrid (A+I)	nested loop, bound	✓	✓
Atlas [50]	DHTF	✓	Java	GNU L.G.P.L	Index only	SQLite	✗	✗
SAFE [52]	SEF	✓	Java	GNU G.P.L	hybrid (A+I)	bind (VENL)	✓	✓
QUETSAL [88]	SEF	✗	Java	GNU G.P.L	hybrid (A+I)	based on Sesame	✓	✓

Query federation over Linked Data does not require the data providers to publish their data as SPARQL endpoints. Instead, the only requirement is that the RDF data follows the Linked Data principles. A downside of these approaches is that they are less time-efficient than the previous approaches due to the URI lookups they perform.

3. query federation on top of distributed hash tables: This type of federation approaches stores RDF data on top of Distributed Hash Tables (DHTs) and use DHT indexing to federate SPARQL queries over multiple RDF nodes. This is a space-efficient solution and can reduce the network cost as well. However, many of the LOD datasets are not stored on top of DHTs.

Each of the above main category can be further divided into three sub-categories (see Table 3):

(a) catalog/index-assisted solutions: These approaches utilize dataset summaries that have been collected in a pre-processing stage. These approaches may lead to more efficient query federation. However, the index needs to be constantly updated to ensure complete results retrieval. The index size should also be kept to a minimum to ensure that it does not significantly increase the overall query processing costs.

(b) catalog/index-free solutions: In these approaches, the query federation is performed without using any stored data summaries. The data source statistics can be collected on-the-fly before the query federation starts. This approach promises that the results retrieved by the engine are complete and up-to-date. However, it may increase the query execution time, depending on the extra processing required for collecting and processing on-the-fly statistics.

(c) hybrid solutions: In these approaches, some of the data source statistics are pre-stored while some are collected on-the-fly, e.g., using SPARQL ASK queries.

Table 3 provides a classification along with the implementation details of the 14 systems which participated in the survey. In addition, we also included QUETSAL and SAFE proposed in this thesis. Note DAW is one of the contribution of this thesis. Overall, we received responses mainly for systems which implemented the SPARQL endpoint federation and hybrid query processing paradigms in Java. Only Atlas [50] implements DHT federation whereas WoDQA [2], LDQPS [54], and SIHJoin [55] implement federation over linked data (LDF). Most of the surveyed systems provides "General Public Licences" with the exception of [54] and [55] which provides "Scala" licence whereas the authors of [11] and [38] have not yet decided which licence type will hold for their tools. Five of the surveyed systems implement

Table 4: Survey outcome: System's features (**R.C.** = Results Completeness, **P.R.R.** = Partial Results Retrieval, **N.B.O.** = No Blocking Operator, **A.Q.P.** = Adaptive Query Processing, **D.D.** = Duplicate Detection, **P.B.Q.P** = Policy-based Query Planning, **Q.R.E.** = Query Runtime Estimation, **Top-K.Q.P** = Top-K query processing, **BOUS** = Based on Underlying System)

Systems	R.C.	P.R.R.	N.B.O / A.Q.P.	D. D.	P.B.Q.P	Provenance	Q.R.E	Top-K.Q.P
FedX	✓	✗	✗	✗	✗	✗	✗	✗
LHD	✗	✗	✗	✗	✗	✗	✗	✗
SPLENDID	✗	✗	✗	✗	✗	✗	✗	✗
FedSearch	✓	✗	✗	✗	✗	✗	✗	✗
GRANATUM	✗	✗	✗	✗	partial	partial	✗	✗
Avalanche	✗	✓	✓	partial	✗	✗	✗	✗
DAW	✗	✓	BOUS	✓	✗	✗	✗	✗
ANAPSID	✗	✗	✓	✗	✗	✗	✗	✗
ADERIS	✗	✗	✓	✗	✗	✗	✗	✗
DARQ	✗	✗	✗	✗	✗	✗	✗	✗
LDQPS	✗	✗	✓	✗	✗	✗	✗	✗
SIHJoin	✗	✗	✓	✗	✗	✗	✗	✗
WoDQA	✓	✗	✗	✗	✗	✗	✗	✗
Atlas	✗	✗	✗	partial	✗	✗	✗	✗
SAFE	✗	✗	✗	✗	✓	✗	✗	✗
QUETSAL	✗	✓	✗	✓	✗	✗	✗	✗

caching mechanisms including [2], [5], [11], [77] and [94]. In addition, both SAFE and QUETSAL also implement caching. Only [1] and [2] provide support for catalog/index update whereas two systems do not require this mechanism by virtue of being index/catalog-free approaches. Our approaches QUETSAL and SAFE also provide support for index update.

Table 4 summarizes the survey outcome w.r.t. different features supported by systems. Only three of the surveyed systems ([5], [94] and QWIDVD) claim that they achieve result completeness and only Avalanche [11] and DAW [77] support partial results retrieval for their implementations. Note that FedX claims result completeness when the cache that it relies on is up-to-date. Since QUETSAL integrates DAW, therefore QUETSAL also support partial results retrieval. Five (i.e., Avalanche, ANAPSID, ADERIS, LDQPS, SIHJoin) of the considered systems support adaptive query processing. Only DAW [77], QUETSAL [88] support duplicate detection whereas DHT and Avalanche [11] claim to support partial duplicate detection. Granatum [38; 39; 49] is the only system that implements privacy and provenance. None of the considered systems implement top-k query processing or query runtime estimation.

Table 5 lists SPARQL clauses supported by the each of 14 systems. GRANATUM and QWIDVD are only two systems that support all of the query constructs used in our survey. It is important to note that most of these query constructs are based on query characteristics defined in BSBM.

Now we discuss other notable federation engines that did not participate in the survey or published after the public survey was conducted. Semantic Web Integrator and Query Engine (SemWIQ) [56], a SPARQL endpoint federation engine using a mediator approach. The SPARQL endpoints need to register first with the mediator using HTTP POST requests with an RDF document attached. The mediator continuously monitors the SPAQL endpoints for any dataset changes and updates the service descriptions automatically. Unlike DARQ, the service descriptions remain up-to-date all time. Hartig et al. [37] present a Linked Data federation that discovers data that might be relevant for answering a query during the query execution itself. The discovery of relevant data is accomplished by traversing RDF links. They use an iterator-based pipeline and a URI prefetching approach for efficient query execution. Same like DAW, Fedra [60] is SPARQL endpoint federation engine for duplicate-aware SPARQL query federation. The main motivation of Fedra is that the index update can be expensive if the underlying endpoints data changes frequently. Thus, they propose an index that can be used to detect duplicate fragments as well as easy to update.

Table 5: Survey outcome: System's Support for SPARQL Query Constructs (**QP**=Query Predicates, **QS**=Query Subjects,**SPL**=SPLENDID, **FedS**=FedSearch, **GRA**=GRANATUM, **Ava**=Avalanche, **ANA**=Query ANAPSID, **ADE**=ADERIS)

SPARQL Cluase	FedX	Atlas	LHD	SPL	FedS	GRA	Ava	DAW	LDQPS	SIHJoin	ANA	ADE	QWIDVD	DARQ	SAFE	QUETSAL
SERVICE	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗	✓	✗	✓	✗	✓	✓
FILTER	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓
Unbound QP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Unbound QS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
OPTIONAL	✓	✗	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓	✓
DISTINCT	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓
ORDER BY	✓	✗	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓	✓
UNION	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓	✓
NEGATION	✓	✗	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓	✓
REGEX	✓	✗	✓	✗	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓
LIMIT	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓
CONSTRUCT	✓	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗
DESCRIBE	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✓
ASK	✓	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓	✓

3.4 DETAILS OF SELECTED SYSTEMS

After having given a general overview of SPARQL query federation systems, we present six SPARQL endpoints federation engines [1; 27; 57; 70; 94; 100] with public implementation that were used within our experiments. Note we compared our proposed approaches with state-of-the-art federation engines in the subsequent chapter. Here, we aim to We begin by presenting an overview of key concepts that underpin federated query processing and are used in the performance evaluation. We then use these key concepts to present the aforementioned six systems used in our evaluation in more detail.

3.4.1 *Overview of the selected approaches*

DARQ [70] makes use of an index known as *service description* to perform *source selection*. Each service description provides a declarative description of the data available in a data source, including the corresponding SPARQL endpoint along with statistical information. The *source selection* is performed by using distinct predicates (for each data source) recorded in the index as *capabilities*. The source selection algorithm used in DARQ for a query simply matches all triple patterns against the *capabilities* of the data sources. The matching compares the predicate in a triple pattern with the predicate defined for a capability in the index. This means that DARQ is only able to answer queries with bound predicates. DARQ combines service descriptions, query rewriting mechanisms and a cost-based optimization approach to reduce the query processing time and the bandwidth usage.

SPLENDID [27] makes use of VoiD descriptions as index along with SPARQL ASK queries to perform the source selection step. A SPARQL ASK query is used when any of the subject or object of the triple pattern is bound. This query is forwarded to all of the data sources and those sources which pass the SPARQL ASK test are selected. A dynamic programming strategy [97] is used to optimize the join order of SPARQL basic graph patterns.

FedX [94] is an index-free SPARQL query federation system. The source selection relies completely on SPARQL ASK queries and a cache. The cache is used to store recent SPARQL ASK operations for relevant data source selection. As shown by our evaluation, the use of this cache greatly reduces the source selection and query execution time.

The publicly available implementation of LHD [100] only makes use of the VoiD descriptions to perform source selection. The source selection algorithm is similar to DARQ. However, it also supports query triple patterns with unbound predicates. In such cases, LHD simply selects all of the available data sources as relevant. This strategy often overestimates the number of capable sources and can thus

lead to high overall runtimes. LHD performs a pipeline hash join to integrate sub-queries in parallel.

ANAPSID [1] is an adaptive query engine that adapts its query execution schedulers to the data availability and runtime conditions of SPARQL endpoints. This framework provides physical SPARQL operators that detect when a source becomes blocked or data traffic is bursty. The operators produce results as quickly as data arrives from the sources. ANAPSID makes use of both a catalog and ASK queries along with heuristics defined in [61] to perform the source selection step. This heuristic-based source selection can greatly reduce the total number of triple pattern-wise selected sources.

Finally, ADERIS [57] is an index-only approach for adaptive integration of data from multiple SPARQL endpoints. The source selection algorithm is similar to DARQ's. However, this framework also selects all of the available data sources for triple patterns with unbound predicates. ADERIS does not support several SPARQL 1.0 clauses such as UNION and OPTIONAL. For the data integration, the framework implements the pipelined index nested loop join operator.

In the next section, we describe known variables that may impact the performance of the federated SPARQL query engines.

3.5 PERFORMANCE VARIABLES

Table 6 shows known variables that may impact the behaviour of federated SPARQL query engines. According to [61], these variables can be grouped into two categories (i.e., independent and dependent variables) that affect the overall performance of federated query SPARQL engines. Dependent (also called observed) variables are usually the performance metrics and are normally influenced by independent variables. Dependent variables include: (1) total number of SPARQL ASK requests used during source selection `#ASK`, (2) total number of triple pattern-wise sources selected during source selection `#TP Sources`, (3) source selection time, (4) overall query runtime, and (5) answer set completeness.

Independent variables can be grouped into four dimensions: query, data, platform, and endpoint [61]. The query dimension includes:

- the type of query (star, path, hybrid [77]),
- the number of basic graph patterns,
- the instantiations (bound/unbound) of tuples (subject, predicate, object) of the query triple pattern,
- the selectivity of the joins between triple patterns,
- the query result set size, and use of different SPARQL clauses along with general predicates such as `rdf:type`, `owl:sameAs`.

Table 6: Known variables that impact the behaviour of SPARQL federated. (#ASK = Total number of SPARQL ASK requests used during source selection, #TP= total triple pattern-wise sources selected, SST = Source Selection Time, QR = Query Runtime, AC = Answer Completeness,)

	Independent Variables	**Dependent/Observed Variables**				
		#ASK	#TP Sources	SST	QR	AC
Query	query plan shape	✓	✓	✓	✓	✓
	#basic triple patterns	✓	✓	✓	✓	✓
	#instantiations and their position	✓	✓	✓	✓	x
	join selectivity	x	x	x	✓	x
	#intermediate results	x	x	x	✓	x
	answer size	x	x	x	✓	x
	usage of query language expressivity	✓	✓	✓	✓	x
	#general predicates	✓	✓	✓	✓	✓
Data	dataset size	x	x	x	✓	x
	data frequency distribution	x	x	x	✓	x
	type of partitioning	✓	✓	✓	✓	✓
	data endpoint distribution	✓	✓	✓	✓	✓
Platform	cache on/off	✓	✓	✓	✓	x
	RAM available	x	x	✓	✓	x
	#processors	x	x	✓	✓	x
Endpoints	#endpoints	✓	✓	✓	✓	✓
	endpoint type	x	x	✓	✓	x
	relation graph/endpoint/instance	x	x	x	✓	✓
	network latency	x	x	✓	✓	✓
	initial delay	x	x	✓	✓	x
	message size	x	x	x	✓	x
	transfer distribution	x	x	✓	✓	✓
	answer size limit	x	x	x	✓	✓
	timeout	x	x	x	✓	✓

The data dimension comprises of:

- the dataset size, its type of partition (horizontal, vertical, hybrid), and
- the data frequency distribution (e.g., number of subject, predicates and objects) etc.

The platform dimension consists of:

- use of cache,
- number of processor, and
- amount of RAM available.

The following parameters belong to the endpoints dimension:

- the number of endpoints used in the federation and their types (e.g., Fuseki, Sesame, Virtuoso etc., and single vs. clustered server),
- the relationship between the number of instances, graphs and endpoints of the systems used during the evaluation, and
- network latency (in case of live SPARQL endpoints) and different endpoint configuration parameters such as answer size limit, maximum resultset size etc.

In our evaluation, we measured all of the five dependent variables reported in Table 6. Most of the query (an independent variable) parameters are covered by using the complete query set of both FedX and SP^{2}Bench. However, as pointed in [61], the join selectivity cannot be fully covered due to the limitations of both FedX and SP^{2}Bench. In data parameters, the data set size cannot be fully explored in the selected SPARQL query federation benchmarks. This is because both FedBench and SP^{2}Bench do not contain very large datasets (the largest dataset in these benchmarks contains solely 108M triples, see Table 8) such as Linked TCGA (20.4 billion triples[13]), UniProt (8.4 billion triples[14]) etc. We used horizontal partitioning and mirrored a highly distributed environment to test the selected federation systems for their parallel processing capabilities. W.r.t. platform parameters, the effect of using a cache is measured. As shown in the experiments section, the use of a cache (especially in FedX) has the potential of greatly improving the query runtime of federation systems. The amount of available RAM is more important when dealing with queries with large intermediate results, which are not given in the benchmarks at hand. The number of processors used is an important dimension to be considered in future SPARQL query federation engines. The endpoint parameters did not play a major role in our

13 Linked TCGA: http://tcga.deri.ie/
14 UniProt: http://datahub.io/dataset/uniprotkb

study because we used a dedicated local area network to avoid network delays. An evaluation on live SPARQL endpoints with network delay will be considered in future work. We used Virtuoso (version 20120802) SPARQL endpoints with maximum rows set to 100,000 (i.e., we chose this value because it is greater than the answer size of all of the queries in the selected benchmarks) and a transaction timeout of 60 seconds (which allows for all all sub-queries in the selected benchmarks to be executed). The overall query execution timeout was set to 30 min on the system running the federation engine. The higher threshold is due to SPARQL endpoints requiring less time to run the sub-queries generated by the federation engine than the federation engine to integrate the results.

While the dependent variables *source selection time*, *query runtime*, and *answer completeness* are already highlighted in [61], we also measured the *total number of data sources selected* and *total number of SPARQL ASK requests* used during the source selection. Section 3.6 shows that both of these additional variables have a significant impact on the performance of federated SPARQL query engines. For example, an overestimation of the capable sources can lead through an increase of the overall runtime due to (1) increased network traffic and (2) unnecessary intermediate results which are excluded after performing all the joins between the query triple patterns. On the other hand, the smaller the number of SPARQL ASK requests used during the source selection, the smaller the source selection time and vice versa. Further details of the depended and independent variables can be found at [61].

3.6 EVALUATION

In this section we present the data and hardware used in our evaluation. Moreover, we explain the key metrics underlying our experiments as well as the corresponding results.

3.6.1 *Experimental setup*

We used two settings to evaluate the selected federation systems. Within the first evaluation, we used the query execution time as central evaluation parameter and made use of the FedBench [92] federated SPARQL querying benchmark. In the second evaluation, we extended both FedBench and SP^{2}Bench to simulate a highly federated environment. Here, we focused especially on analyzing the effect of data partitioning on the performance of federation systems. We call this extension *SlicedBench* as we created slices of each original datasets and distributed them among data sources. All of the selected performance metrics (explained in Section 3.6.2) remained the same for both evaluation frameworks. We used the most recent versions (at the time

at which the evaluation was carried out), i.e., FedX2.0 and ANAPSID (December 2013 version). The remaining systems has no versions. All experiments were carried out on a system (machine running federation engines) with a 2.60 GHz i5 processor, 4GB RAM and 500GB hard disk. For systems with Java implementation, we used Eclipse with default settings, i.e., Java Virtual Machine (JVM) initial memory allocation pool (Xms) size of 40MB and the maximum memory allocation pool (Xmx) size of 512MB. The permanents generation (MaxPermSize) which defines the memory allocated to keep compiled class files was also set to default size of 256MB. To minimise the network latency we used a dedicated local network. We conducted our experiments on local copies of Virtuoso (version 20120802) SPARQL endpoints with number of buffers 1360000, maximum dirty buffers 1000000, number of server threads 20, result set maximum rows 100000, and maximum SPARQL endpoint query execution time of 60 seconds. A separate physical virtuoso server was created for each dataset. The specification of the machines hosting the virtuoso SPARQL endpoints used in both evaluations is given in Table 7. We executed each query 10 times and present the average values in the results. The source selection time (ref. Section 3.6.3.4) and query runtime (ref. Section 3.6.3.5) was calculated using the function `System.currentTimeMillis()` (for Java system implementations) and function `time()` (for Python implementations). The results of the `time()` was converted from seconds as float to milliseconds. The accuracy of both functions is in the order of 1ms, which does not influence the conclusions reached by our evaluation. The query runtime was calculated once all the results are retrieved and the time out was set to 30 minutes. Furthermore, the query runtime results were analyzed statistically using Wilcoxon signed rank test. We chose this test because it is parameter-free and does not assume a particular error distribution in the data like a t-test does. For all the significance tests, we set the p-value to 0.05.

All of the data used in both evaluations along with the portable virtuoso SPARQL endpoints can be downloaded from the project website[15].

3.6.1.1 *First setting: FedBench*

FedBench is commonly used to evaluate performance of the SPARQL query federation systems [27; 61; 77; 94]. The benchmark is explicitly designed to represent SPARQL query federation on a real-world datasets. The datasets can be varied according to several dimensions such as size, diversity and number of interlinks. The benchmark queries resemble typical requests on these datasets and their structure ranges from simple star [77] and chain queries to complex graph patterns. The details about the FedBench datasets used in our evaluation along with some statistical information are given in Table 8.

15 https://code.google.com/p/fed-eval/

Table 7: System's specifications hosting SPARQL endpoints.

Endpoint	CPU(GHz)	RAM	Hard Disk
SW Dog Food	2.2, i3	4GB	300 GB
GeoNames	2.9, i7	16 GB	256 GB SSD
KEGG	2.6, i5	4 GB	150 GB
Jamendo	2.53, i5	4 GB	300 GB
New York Times	2.3, i5	4 GB	500 GB
Drugbank	2.53, i5	4 GB	300 GB
ChEBI	2.9, i7	8 GB	450 GB
LinkedMDB	2.6, i5	8 GB	400 GB
SP^2Bench	2.6, i5	8 GB	400 GB
DBpedia subset 3.5.1	2.9, i7	16 GB	500 GB

The queries included in FedBench are divided into three categories: Cross-domain (CD), Life Sciences (LS), Linked Data (LD). In addition, it includes SP^2Bench queries. The distribution of the queries along with the result set sizes are given in Table 9. Details on the datasets and various advanced statistics are provided at the FedBench project page[16].

16 http://code.google.com/p/fbench/

Table 8: Datasets statistics used in our benchmarks. (*only used in SlicedBench)

Collection	Dataset	version	#triples	#subjects	#predicates	#objects	#types	#links
Cross Domain	DBpedia subset	3.5.1	43.6M	9.50M	1.06k	13.6M	248	61.5k
	GeoNames	2010-10-06	108M	7.48M	26	35.8M	1	118k
	LinkedMDB	2010-01-19	6.15M	694k	222	2.05M	53	63.1k
	Jamendo	2010-11-25	1.05M	336k	26	441k	11	1.7k
	New York Times	2010-01-13	335k	21.7k	36	192k	2	31.7k
	SW Dog Food	2010-11-25	104k	12.0k	118	37.5k	103	1.6k
Life Sciences	KEGG	2010-11-25	1.09M	34.3k	21	939k	4	30k
	ChEBI	2010-11-25	7.33M	50.5k	28	772k	1	-
	Drugbank	2010-11-25	767k	19.7k	119	276k	8	9.5k
	DBpedia subset 3.5.1	43.6M	9.50M	1.06k	13.6M	248	61.5k	61.5k
SP^{2}Bench*	SP^{2}Bench 10M	v1.01	10M	1.7M	77	5.4M	12	-

In this evaluation setting, we selected all queries from CD, LS, and LD, thus performing (to the best of our knowledge) the first evaluation of SPARQL query federation systems on the complete benchmark data of FedBench. It is important to note that SP^{2}Bench was designed with the main goal of evaluating query engines that access data kept in a single repository. Thus, the complete query is answered by a single data set. However, a federated query is one which collects results from multiple data sets. Due to this reason we did not include the SP^{2}Bench queries in our first evaluation. We have included all these queries into our SlicedBench because the data is distributed in 10 different data sets and each SP^{2}Bench query span over more than one data set, thus full-filling the criteria of federated query.

Table 9: Query characteristics, (#T = Total number of Triple patterns, #Res = Total number of query results, *only used in SlicedBench, **OP**erators: **A**nd (“.”), **U**nion, **F**ilter, **O**ptional; **Struc**ture: **S**tar, **C**hain, **H**ybrid).

Linked Data (LD)					Cross Domain (CD)					Life Science (LS)					SP^{2}Bench*				
Q	#T	#Res	Op	Struct		#T	#Res	Op	Struct	Q	#T	#Res	Op	Struct	Q	#T	#Res	Op	Struc
1	3	309	A	C	1	3	90	AU	S	1	2	1159	U	-	1	3	1	A	S
2	3	185	A	C	2	3	1	A	S	2	3	333	AU	-	2	10	>50k	AO	S
3	4	162	A	C	3	5	2	A	H	3	5	9054	A	H	3a	3	27789	AF	S
4	5	50	A	C	4	5	1	A	C	4	7	3	A	H	4	8	>40k	AF	C
5	3	10	A	S	5	4	2	A	C	5	6	393	A	H	5b	5	>30k	AF	C
6	5	11	A	H	6	4	11	A	C	6	5	28	A	H	6	9	>70k	AFO	H
7	2	1024	A	S	7	4	1	A	C	7	5	144	AFO	H	7	12	>2k	AFO	H
8	5	22	A	H											8	10	493	AFU	H
9	3	1	A	C											9	4	4	AU	-
10	3	3	A	C											10	1	656		-
11	5	239	A	S											11	1	10	-	-

3.6.1.2 *Second setting: Sliced Bench*

As pointed out in [62] the data partitioning can affect the overall performance of SPARQL query federation engines. To quantify this effect, we created 10 slices of each of the 10 datasets given in Table 8 and distributed this data across 10 local virtuoso SPARQL endpoints (one slice per SPARQL endpoint). Thus, every SPARQL endpoint contained one slice from each of the 10 datasets. This creates a highly fragmented data environment where a federated query possibly had to collect data from all of the 10 SPARQL endpoints. This characteristic of the benchmark stands in contrast to FedBench where the data is not highly fragmented. Moreover, each of the SPARQL endpoint contained a comparable amount of triples (load balancing). To facilitate the distribution of the data, we used the Slice Generator tool employed in [77]. This tool allows setting a discrepancy across the slices, where the *discrepancy* is defined as the difference (in terms of number of triples) between the largest and smallest slice:

$$\mathtt{discrepancy} = \max_{1 \leqslant i \leqslant M} |S_i| - \min_{1 \leqslant j \leqslant M} |S_j|, \tag{1}$$

where S_i stands for the i^{th} slice. The dataset D is partitioned randomly among the slices in a way that $\sum_i |S_i| = |D|$ and $\forall i \forall j \, i \neq j \rightarrow ||S_i| - |S_j|| \leqslant \mathtt{discrepancy}$.

This tool generate slices based on horizental partioning of the data. Table 10 shows the discrepancy values used for slice generation for each of the 10 datasets. Our discrepancy value varies with the size of the dataset. For the query runtime evaluation, we selected all of the queries both from FedBench and SP^2Bench given in Table 9: the reason for this selection was to cover majority of the SPARQL query clauses and types along with variable results size (from 1 to 40 million). For each of the CD, LS, and LD queries used in SlicedBench, the number of results remained the same as given in Table 9. Analogously to FedBench, each of the SlicedBench data source is a virtuoso SPARQL endpoint.

3.6.2 *Evaluation criteria*

We selected five metrics for our evaluation: (1) *total triple pattern-wise sources selected*, (2) total number of SPARQL ASK requests used during source selection, (3) *answer completeness* (4) *source selection time* (i.e. the time taken by the process in the first metric), and (5) *query execution time*.

As an example, consider we have a query containing two triple patterns. Let there are three sources capable of answering the first triple pattern and four sources capable of answering summing up to a total triple pattern-wise selected sources equal to seven.

Table 10: Dataset slices used in SlicedBench

Collection	#Slices	Discrepancy
DBpedia subset 3.5.1	10	280000
GeoNames	10	600000
LinkedMDB	10	100000
Jamendo	10	30000
New York Times	10	700
SW Dog Food	10	200
KEGG	10	35000
ChEBI	10	50000
Drugbank	10	25000
SP^2Bench	10	150000

An overestimation of triple pattern-wise selected sources increases the source selection time and thus the the query execution time. Furthermore, such an overestimation increases the number of irrelevant results which are excluded after joining the results of the different sources, therewith increasing both the network traffic and query execution time. In the next section we explain how such overestimations occur in the selected approaches.

3.6.3 *Experimental results*

3.6.3.1 *Triple pattern-wise selected sources*

Table 11 shows the total number of triple pattern-wise sources (TP sources for short) selected by each approach both for the FedBench and SlicedBench queries. ANAPSID is the most accurate system in terms of TP sources followed by both FedX and SPLENDID whereas similar results are achieved by the other three systems, i.e., LHD, DARQ, and ADERIS. Both FedX and SPLENDID select the optimal number of TP sources for individual query triple patterns. This is because both make use of ASK queries when any of the subject or object is bound in a triple pattern. However, they do not consider whether a source can actually contribute results after performing a join between results with other query triple patterns. Therefore, both can overestimate the set of capable sources that can actually contribute results. ANAPSID uses a catalog and ASK queries along with heuristics [61] about triple pattern joins to reduce the overestimation of sources. LHD (the publicly available version), DARQ, and ADERIS are index-only approaches and do not use SPARQL ASK queries when any of the subject or object is bound. Consequently, these three approaches tend to overestimate the TP sources per individual triple pattern. It

Table 11: Comparison of triple pattern-wise total number of sources selected for FedBench and SlicedBench. NS stands for "not supported", RE for "runtime error", SPL for SPLENDID, ANA for ANAPSID and ADE for ADERIS. Key results are in **bold**.

FedBench							SlicedBench						
Query	FedX	SPL	LHD	DARQ	ANA	ADE	Query	FedX	SPL	LHD	DARQ	ANA	ADE
CD1	11	11	**28**	NS	3	NS	CD1	17	17	**30**	NS	8	NS
CD2	3	3	10	10	3	10	CD2	12	12	24	24	12	24
CD3	**12**	**12**	**20**	**20**	**5**	**20**	CD3	31	31	38	38	31	38
CD4	19	19	20	20	5	20	CD4	32	32	34	34	32	34
CD5	11	11	11	11	4	11	CD5	19	19	19	19	9	19
CD6	**9**	**9**	**10**	**10**	**10**	**10**	CD6	31	31	40	40	31	40
CD7	13	13	13	13	6	13	CD7	40	40	40	40	40	40
Total	**78**	**78**	**112**	**84**	**36**	**84**	**Total**	**182**	**182**	**225**	**195**	**163**	**195**
LS1	**1**	**1**	**1**	**1**	**1**	NS	LS1	3	3	3	3	3	NS
LS2	**11**	**11**	**28**	NS	**12**	NS	LS2	16	16	**30**	NS	16	NS
LS3	12	12	20	20	5	20	LS3	19	19	26	26	19	26
LS4	7	7	15	15	7	15	LS4	25	25	27	27	14	27
LS5	10	10	18	18	7	18	LS5	30	30	37	37	20	37
LS6	9	9	17	17	5	17	LS6	19	19	27	27	17	27
LS7	6	6	6	6	7	NS	LS7	13	13	13	13	13	NS
Total	**56**	**56**	**105**	**77**	**44**	**70**	**Total**	**125**	**125**	**163**	**133**	**102**	**117**
LD1	8	8	11	11	3	11	LD1	10	10	29	29	3	29
LD2	3	3	3	3	3	3	LD2	20	20	28	28	20	28
LD3	16	16	16	16	4	16	LD3	30	30	39	39	13	39
LD4	5	5	5	5	5	5	LD4	30	30	47	47	5	47
LD5	5	5	13	13	3	13	LD5	15	15	24	24	15	24
LD6	14	14	14	14	14	14	LD6	38	38	38	38	38	38
LD7	3	3	4	4	2	4	LD7	12	12	20	20	12	20
LD8	15	15	15	15	9	15	LD8	27	27	27	27	16	27
LD9	3	3	6	6	3	6	LD9	7	7	17	17	7	17
LD10	10	10	11	11	3	11	LD10	23	23	23	23	23	23
LD11	15	15	15	15	5	15	LD11	31	31	32	32	31	32
Total	**108**	**108**	**119**	**122**	**54**	**119**	**Total**	**243**	**243**	**324**	**324**	**183**	**324**
							SP2B-1	10	10	28	28	NS	28
							SP2B-2	90	90	92	92	RE	NS
							SP2B-3a	13	13	19	NS	13	19
							SP2B-4	52	52	66	66	52	66
							SP2B-5b	40	40	50	50	40	50
							SP2B-6	68	68	72	72	18	NS
							SP2B-7	100	100	104	NS	64	NS
							SP2B-8	91	91	102	102	NS	NS
							SP2B-9	40	40	40	NS	40	NS
							SP2B-10	7	7	10	NS	7	10
							SP2B-11	10	10	10	10	10	NS
							Total	**521**	**521**	**593**	**420**	**244**	**173**
Net Total	**242**	**242**	**336**	**283**	**134**	**273**	**Net Total**	**1071**	**1071**	**1305**	**1072**	**692**	**809**

is important to note that DARQ does not support queries where any of the predicates in a triple pattern is unbound (e.g., CD1, LS2) and ADERIS does not support queries which feature FILTER or UNION clauses (e.g., CD1, LS1, LS2, LS7). In case of triple patterns with unbound predicates (such as CD1, LS2) both LHD and ADERIS simply select all of the available sources as relevant. This overestimation can significantly increase the overall query execution time.

The effect overestimation can be clearly seen by taking a fine-granular look at how the different systems process FedBench query CD3 given in Listing 4. The optimal number of TP sources for this query is 5. This query has a total of five triple patterns. To process this query, FedX sends a SPARQL ASK query to all of the 10 benchmark SPARQL endpoints for each of the triple pattern summing up to a total of 50 (5*10) SPARQL ASK operations. As a result of these operations, only one source is selected for each of the first four triple pattern while eight sources are selected for last one, summing up to a total of 12 TP sources. SPLENDID utilizes its index and ASK queries for the first three and index-only for last two triple pattern to select exactly the same number of sources selected by FedX. LHD, ADERIS, and DARQ only makes use of predicate lookups in their catalogs to select nine sources for the first, one source each for the second, third, fourth, and eighth for the last triple pattern summing up to a total of 20 TP sources. The later three approaches overestimate the number of sources for first triple pattern by 8 sources. This is due to the predicate `rdf:type` being likely to be used in all of RDF datasets. However, triples with `rdf:type` as predicate and the bound object `dbp:President` are only contained in the DBpedia subset of FedBench. Thus, the only relevant data source for the first triple pattern is DBpedia subset. Interestingly, even FedX and SPLENDID overestimate the number of data sources that can contribute for the last triple pattern. There are eight FedBench datasets which contain `owl:sameAs` predicate. However, only one (i.e., New York Times) can actually contribute results after a join of the last two triple patterns is carried out. ANAPSID makes use of a catalog and SPARQL-ASK-assisted Star Shaped Group Multiple (SSGM) endpoint selection heuristic [61] to select the optimal (i.e., five) TP sources for this query. However, ANAPSID also overestimates the TP sources in some cases. For query CD6 of FedBench, ANAPSID selected a total of 10 TP sources while only 4 is the optimal sources that actually contributes to the final result set. This behaviour leads us to our first insight: Optimal TP source selection is not sufficient to detect the optimal set of sources that should be queried.

In the SlicedBench results, we can clearly see the TP values are increased for each of the FedBench queries which mean a query spans more data sources, thus simulating a highly fragmented environment suitable to test the federation system for effective parallel query pro-

```
SELECT ?president ?party ?page
WHERE {
?president rdf:type dbp:President .
?president dbp:nationality dbp:US .
?president dbp:party ?party .
?x nyt:topicPage ?page .
?x owl#sameAs ?president .
}
```

Listing 4: FedBench CD3. Prefixes are ignored for simplicity

cessing. The highest number of TP sources are reported for the second SP^{2}Bench query where up to a total of 92 TP sources are selected. This query contains 10 triple patterns and index-free approaches (e.g., FedX) need 100 (10*10) SPARQL ASK queries to perform the source selection operation. Using SPARQL ASK queries with no caching for such a highly federated environment can be very expensive. From the results shown in Table 11, it is noticeable that hybrid (catalog + SPARQL ASK) source selection approaches (ANAPSID, SPLENDID) perform an more accurate source selection than index/catalog-only approaches (i.e., DARQ, LHD, and ADERIS).

3.6.3.2 *Number of SPARQL ASK requests*

Table 12 shows the total number of SPARQL ASK requests used to perform source selection for each of the queries of FedBench and SlicedBench. Index-only approaches (DARQ, ADERIS, LHD) only make use of their index to perform source selection. Therefore, they do not necessitate any ASK requests to process queries. As mention before, FedX only makes use of ASK requests (along with a cache) to perform source selection. The results presented in Table 12 are for FedX(cold or first run), where the FedX cache is empty. This is basically the lower bound of the performance of FedX. For FedX(100% cached), the complete source selection is performed by using cache entries only. Hence, in that case, the number of SPARQL ASK requests is zero for each query. This is the upper bound of the performance of FedX on the data at hand. The results clearly shows that index-free (e.g., FedX) approaches can be very expensive in terms of SPARQL ASK requests used. This can greatly affect the source selection time and overall query execution time if no cache is used. Both for FedBench and SlicedBench, SPLENDID is the most efficient hybrid approach in terms of SPARQL ASK requests consumed during source selection.

For SlicedBench, all data sources are likely contains the same set of distinct predicates (because each data source contains at least one slice from each data dump). Therefore, the index-free and hybrid source selection approaches are bound to consume more SPARQL ASK requests. It is important to note that ANAPSID combines more

Table 12: Comparison of number of SPARQL ASK requests used for source selection both in FedBench and SlicedBench. NS stands for "not supported", RE for "runtime error", SPL for SPLENDID, ANA for ANAPSID and ADE for ADERIS. Key results are in **bold**.

	FedBench							SlicedBench					
Query	FedX	SPL	LHD	DARQ	ANA	ADE	Query	FedX	SPL	LHD	DARQ	ANA	ADE
CD1	27	26	0	NS	20	NS	CD1	30	30	0	NS	25	NS
CD2	27	9	0	0	1	0	CD2	30	20	0	0	29	0
CD3	45	2	0	0	2	0	CD3	50	20	0	0	46	0
CD4	45	2	0	0	3	0	CD4	50	10	0	0	34	0
CD5	36	1	0	0	1	0	CD5	40	10	0	0	14	0
CD6	36	2	0	0	11	0	CD6	40	10	0	0	40	0
CD7	36	2	0	0	5	0	CD7	40	10	0	0	40	0
Total	**252**	**44**	**0**	**0**	**43**	**0**	**Total**	**280**	**110**	**0**	**0**	**228**	**0**
LS1	18	0	0	0	0	NS	LS1	20	0	0	0	3	NS
LS2	27	26	0	NS	30	NS	LS2	30	30	0	NS	30	NS
LS3	45	1	0	0	13	0	LS3	50	10	0	0	30	0
LS4	63	2	0	0	1	0	LS4	70	20	0	0	15	0
LS5	54	1	0	0	4	0	LS5	60	10	0	0	27	0
LS6	45	2	0	0	13	0	LS6	50	20	0	0	26	0
LS7	45	1	0	0	2	NS	LS7	50	10	0	0	12	NS
Total	**297**	**33**	**0**	**0**	**63**	**0**	**Total**	**330**	**100**	**0**	**0**	**143**	**0**
LD1	27	1	0	0	1	0	LD1	30	10	0	0	12	0
LD2	27	1	0	0	0	0	LD2	30	10	0	0	29	0
LD3	36	1	0	0	2	0	LD3	40	10	0	0	23	0
LD4	45	2	0	0	0	0	LD4	50	20	0	0	25	0
LD5	27	2	0	0	2	0	LD5	30	20	0	0	32	0
LD6	45	1	0	0	12	0	LD6	50	10	0	0	38	0
LD7	18	2	0	0	4	0	LD7	20	10	0	0	20	0
LD8	45	1	0	0	7	0	LD8	50	10	0	0	19	0
LD9	27	5	0	0	3	0	LD9	30	20	0	0	17	0
LD10	27	2	0	0	4	0	LD10	30	10	0	0	23	0
LD11	45	1	0	0	2	0	LD11	50	10	0	0	32	0
Total	**369**	**19**	**0**	**0**	**37**	**0**	**Total**	**410**	**140**	**0**	**0**	**270**	**0**
							SP2B-1	30	20	0	0	NS	0
							SP2B-2	100	10	0	0	RE	NS
							SP2B-3a	20	10	0	NS	10	0
							SP2B-4	80	20	0	0	66	0
							SP2B-5b	50	20	0	0	50	0
							SP2B-6	90	20	0	0	37	NS
							SP2B-7	130	30	0	NS	62	NS
							SP2B-8	100	20	0	0	NS	NS
							SP2B-9	40	20	0	NS	20	NS
							SP2B-10	10	10	0	NS	10	0
							SP2B-11	10	0	0	0	10	NS
							Total	**660**	**180**	**0**	**0**	**265**	**0**
Net Total	**918**	**96**	**0**	**0**	**143**	**0**	**Net Total**	**1680**	**530**	**0**	**0**	**906**	**0**

than one triple pattern into a single SPARQL ASK query. The time required to execute these more complex SPARQL ASK operations are generally higher than SPARQL ASK queries having a single triple pattern as used in FedX and SPLENDID. Consequently, even though ANAPSID require less SPARQL ASK requests for many of the Fed-Bench queries, its source selection time is greater than all other selected approaches. This behaviour will be further elaborated upon in the subsequent section. Tables 11 and 12 clearly show that using SPARQL ASK queries for source selection leads to an efficient source selection in terms of TP sources selected. However, in the next section we will see that they increase both source selection and overall query runtime. A smart source selection approach should select fewer number of TP sources while using minimal number of SPARQL ASK requests.

3.6.3.3 *Answer compeleteness*

As pointed in [61], an important criterion in performance evaluation of the federated SPARQL query engines is the result set completeness. Two or more engines are only comparable to each other if they provide the same result set for a given query. A federated engine may miss results due to various reasons including the type of source selection used, the use of an outdated cache or index, the type of network, the endpoint result size limit or even the join implementation. In our case, the sole possible reason for missing results across all six engines is the join implementation as all of the selected engines overestimate the set of capable sources (i.e., they never generate false negatives w.r.t. the capable sources), the cache, index are always up-to-date, the endpoint result size limit is greater than the query results and we used a local dedicated network with negligible network delay. Table 13 shows the queries and federated engines for which we did not receive the complete results. As an overall answer completeness evaluation, only FedX is always able to retrieve complete results. It is important to note that these results are directly connected to the answer completeness results presented in survey Table 4; which shows only FedX is able to provide complete results among the selected systems.

Table 13: The queries for which some system's did not retrieve complete results. The values inside bracket shows the actual results size. "-" means the results completeness cannot be determined due to query execution timed out. Incomplete results are highlighted in bold

	CD1(90)	CD7(1)	LS1(1159)	LS2(333)	LS3(9054)	LS5(393)	LD1(309)	LD3(162)	LD9(1)	SP2B-3a(27789)	SP2B-6(>70k)
SPLENDID	90	1	1159	333	9054	393	**308**	**159**	1	-	-
LHD	**77**	1	**0**	**322**	**0**	**0**	309	162	1	-	-
ANAPSID	90	**0**	1159	333	9054	393	309	162	1	**0**	**0**
ADERIS	**77**	1	1159	333	9054	393	309	162	1	-	-
DARQ	90	1	1159	333	9054	393	309	162	**0**	-	-
FedX	90	1	1159	333	9054	393	309	162	1	27789	-

3.6.3.4 *Source selection time*

Figure 6 to Figure 12 show the source selection time for each of the selected approaches and for both FedBench and SlicedBench. Compared to the TP results, the index-only approaches require less time than the hybrid approaches even though they overestimated the TP sources in comparison with the hybrid approaches. This is due to index-only approaches not having to send any SPARQL ASK queries during the source selection process. The index being usually preloaded into the memory before the query execution means that the runtime the predicate look-up in index-only approaches is minimal. Consequently, we observe a trade-off between the intelligent source selection and the time required to perform this process. To reduce the costs associated with ASK operations, FedX implements a cache to store the results of the recent SPARQL ASK operations. Our source selection evaluation results show that source selection time of FedX with cached entries is significantly smaller than FedX's first run with no cached entries.

As expected the source selection time for FedBench queries is smaller than that for SlicedBench, particularly in hybrid approaches. This is because the number of TP sources for SlicedBench queries are increased due to data partitioning. Consequently, the number of SPARQL ASK requests grows and increases the overall source selection time. As mentioned before, an overestimation of TP sources in highly federated environments can greatly increase the source selection time. For example, consider query LD4. SPLENDID selects the optimal (i.e., five) number of sources for FedBench and the source selection time is 218 ms. However, it overestimates the number of TP sources for SlicedBench by selecting 30 instead of 5 sources. As a result, the source selection time is significantly increased to 1035 ms which directly affects the overall query runtime. The effect of such overestimation is even worse in SP2B-2 and SP2B-4 queries for the SlicedBench.

Lessons learned from the evaluation of the first three metrics is that using ASK queries for source selection leads to smart source selection in term of total TP sources selected. On the other hand, they significantly increase the overall query runtime where no caching is used. FedX makes use of an intelligent combination of parallel ASK query processing and caching to perform the source selection process. This parallel execution of SPARQL ASK queries is more time-efficient than the ASK query processing approaches implemented in both ANAPSID and SPLENDID. Nevertheless, the source selection of FedX could be improved further by using heuristics such as ANAPSID's to reduce the overestimation of TP sources.

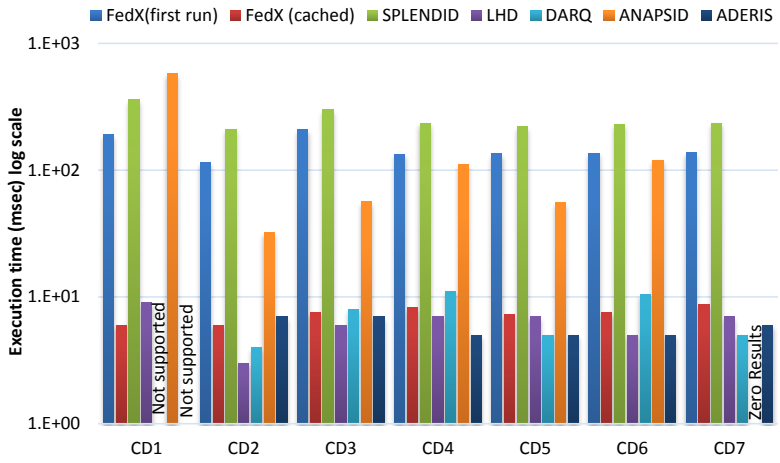


Figure 6: Comparison of source selection time: FedBench CD queries

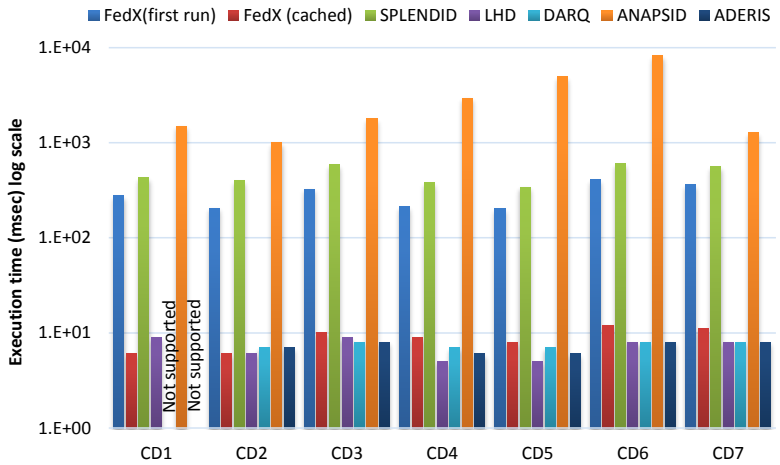


Figure 7: Comparison of source selection time: SlicedBench CD queries

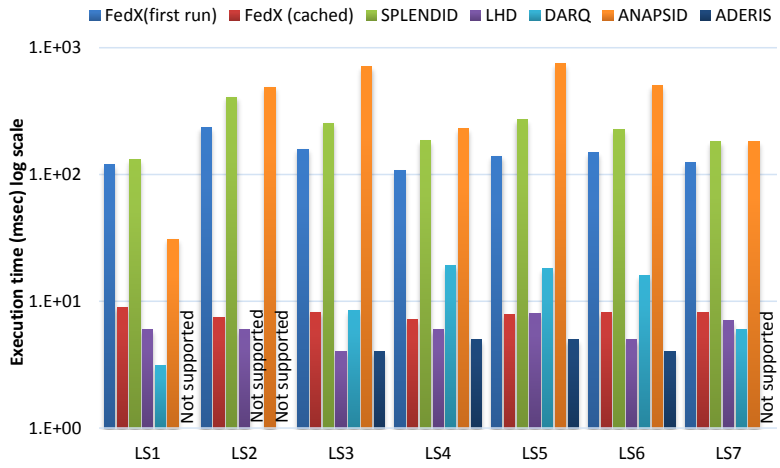


Figure 8: Comparison of source selection time: FedBench LS queries

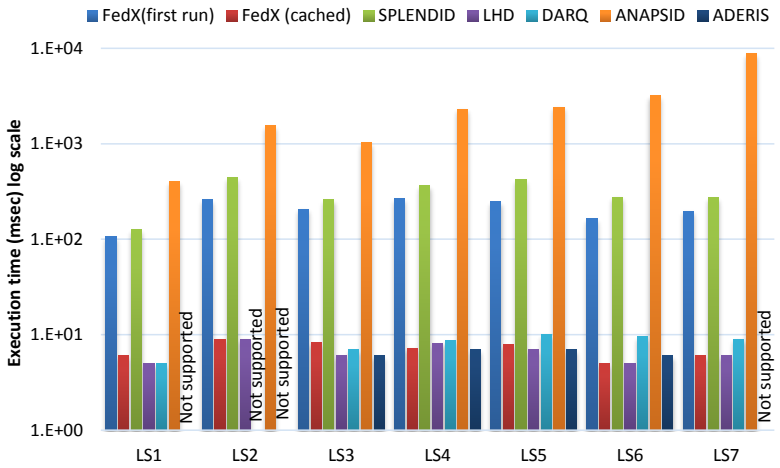


Figure 9: Comparison of source selection time: SlicedBench LS queries

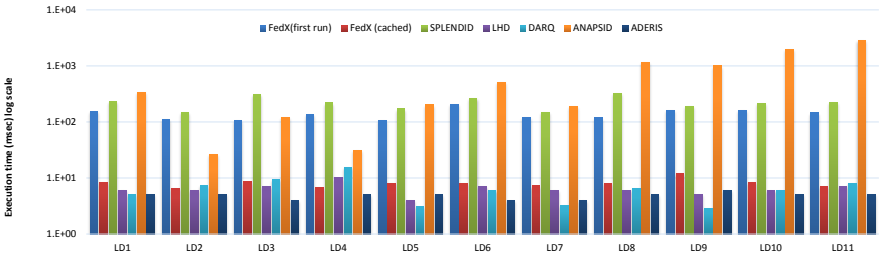


Figure 10: Comparison of source selection time: FedBench LD queries

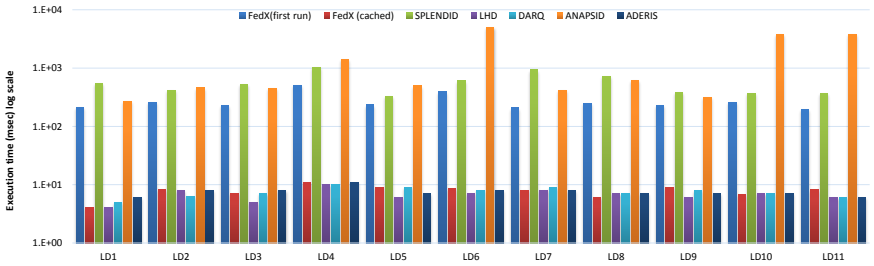


Figure 11: Comparison of source selection time: SlicedBench LD queries

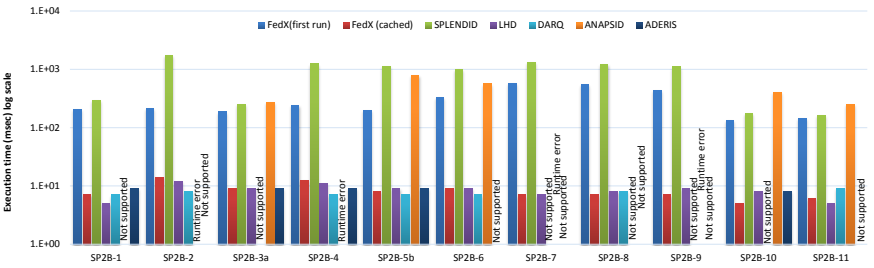


Figure 12: Comparison of source selection time: SlicedBench SP^{2}Bench queries.

3.6.3.5 *Query execution time*

Figure 13 to Figure 19 show the query execution time for both experimental setups. The negligibly small standard deviation error bars (shown on top of each bar) indicate that the data points tend to be very close to the mean, thus suggest a high consistency of the query runtimes in most frameworks. As an overall query execution time evaluation, FedX(cached) outperforms all of the remaining approaches in majority of the queries. FedX(cached) is followed by FedX(first run) which is further followed by LHD, SPLENDID, ANAPSID, ADERIS, and DARQ. Deciding between DARQ and ADERIS is not trivial because the latter does not produce results for most of the queries. The exact number of queries by which one system is better than other is given in the next section (ref. Section 3.7.1). Furthermore, the number of queries by which one system significantly outperform other (using Wilcoxon signed rank test) is also given in the next section.

Interestingly, while ANAPSID ranks first (among the selected systems) in terms of triple pattern-wise sources selected results, it ranks fourth in terms of query execution performance. There are a couple of reason for this: (1) ANAPSID does not make use of cache. As a result, it spends more time (8ms for FedX and 1265 ms for ANAPSID on average over both setups) performing source selection, which worsens its query execution time and (2) Bushy tress (used in ANAPSID) only perform better than left deep trees (used in FedX) when the queries are more complex and triple patterns joins are more selective [43; 4]. However, the FedBench queries (excluding SP^{2}Bench) are not very selective and are rather simple, e.g., triple patterns in a query ranges from 2 to 7. In addition, the query result set sizes are small (10 queries whose resultset size smaller than 16) and the average query execution is small (about 3 seconds on average for FedX over both setups). The SP^{2}Bench queries are more complex and the resultset sizes are large. However, the selected systems were not able to execute majority of the SP^{2}Bench queries. It would be interesting to compare these systems on more complex and Big Data benchmark. The use of a cache improves FedX's performance by 10.5% in the average query execution for FedBech and 4.14% in SlicedBench.

The effect of the overestimation of the TP sources on query execution can be observed on the majority of the queries for different systems. For instance, for FedBench's LD4 query SPLENDID selects the optimal number of TP sources (i.e., five) and the query execution time is 318 ms of which 218 ms are used for selecting sources. For SlicedBench, SPLENDID overestimates the TP sources by 25 (i.e., selects 30 instead of 5 sources), resulting in a query execution of 10693 ms, of which 1035 ms are spent in the source selection process. Consequently, the pure query execution time of this query is only 100 ms for FedBench (318-218) and 9659 ms (10693-1035) for SlicedBench. This means that an overestimation of TP sources does not only in-

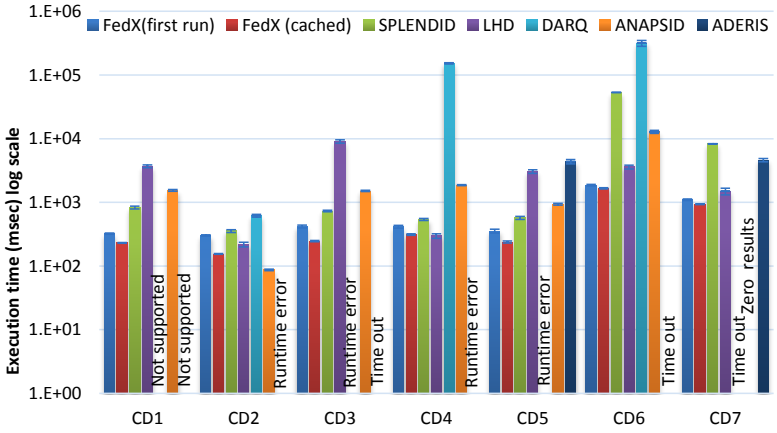


Figure 13: Comparison of query execution time: FedBench CD queries

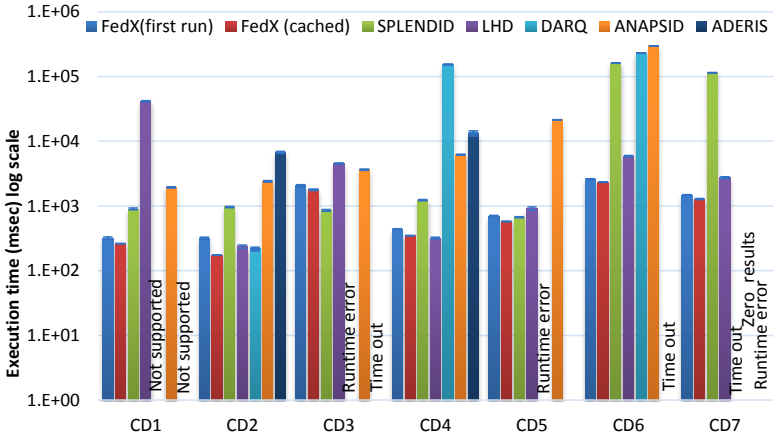


Figure 14: Comparison of query execution time: SlicedBench CD queries

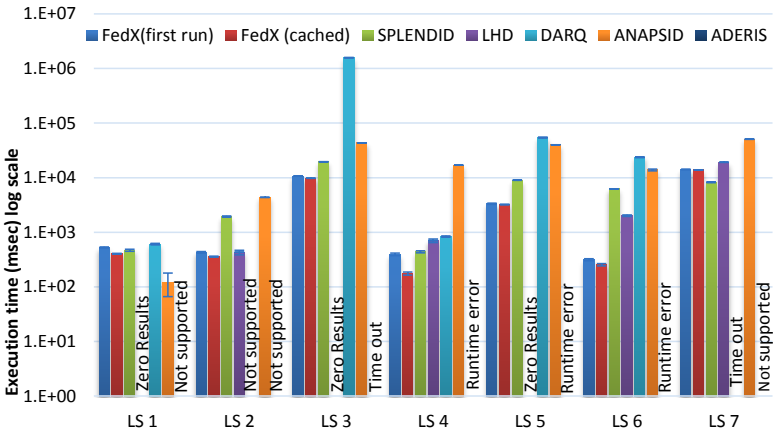


Figure 15: Comparison of query execution time: FedBench LS queries

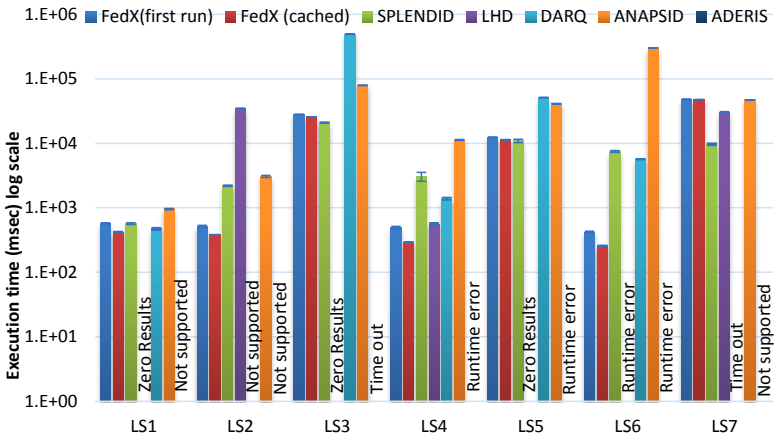


Figure 16: Comparison of query execution time: SlicedBench LS queries

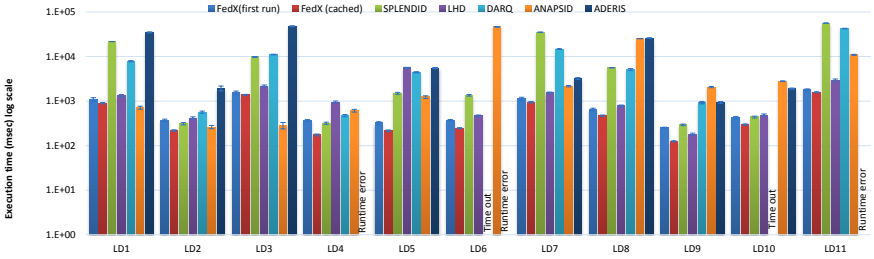


Figure 17: Comparison of query execution time: FedBench LD queries

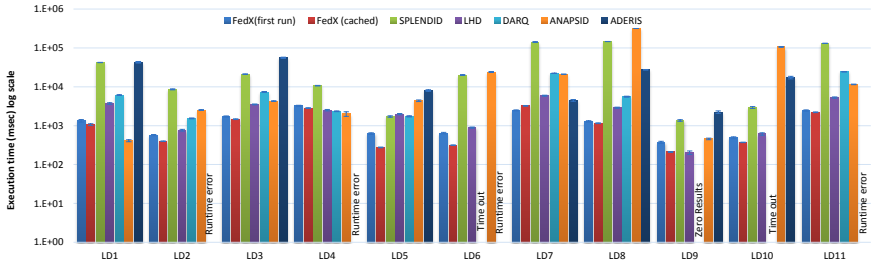
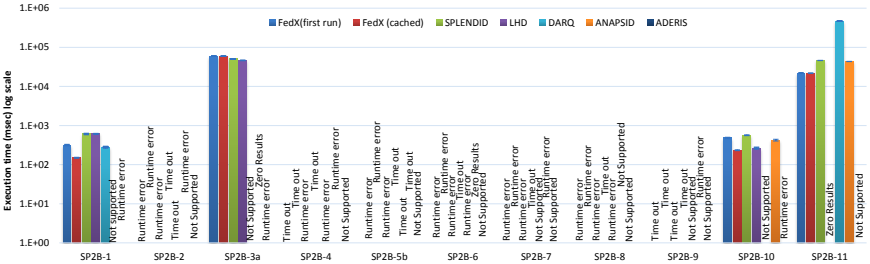


Figure 18: Comparison of query execution time: SlicedBench LD queries

Figure 19: Comparison of query execution time: SlicedBench SP^{2}Bench queries



crease the source selection time but also produces results which are excluded after performing join operation between query triple patterns. These retrieval of irrelevant results increases the network traffic and thwarts the query execution plan. For example, both FedX and SPLENDID considered 285412 irrelevant triples due to the overestimation of 8 TP sources only for $owl:sameAs$ predicate in CD3 of FedBench. Another example of TP source overestimation can seen in CD1, LS2. LHD's overestimation of TP sources on SlicedBench (e.g., 22 for CD1, 14 for LS2) leads to its query execution time jumping from 3670.9 ms to 41586.3 ms for CD1 and 427 ms to 34418.3 ms for LS2.

In queries such as CD4, CD6, LS3, LD11 and SP2B-11 we observe that the query execution time for DARQ is more than 2 minutes. In some cases, it even reaches the 30 minute timeout used in our experiments. The reason for this behaviour is that the simple nested loop join it implements overfloods SPARQL endpoints by submitting too many endpoint requests. FedX overcomes this problem by using a block nested loop join where the number of endpoints requests are dependent upon the block size. Furthermore, we can see that many systems do not produce results for SP^{2}Bench queries. A possible reason for this is the fact that SP^{2}Bench queries contain up to 10 triple patterns with different SPARQL clauses such as DISTINCT, ORDER BY, and complex FILTERS.

3.6.3.6 *Overall performance evaluation*

The comparison of the overall performance of each approach is summarised in Figure 20, where we show the average query execution time for the queries in CD, LS, LD, and SP^{2}Bench sub-groups. As an overall performance evaluation based on FedBench, FedX(cached) outperformed FedX(first run) on all of the 25 queries. FedX(first run) in turn outperformed LHD on 17 out of 22 commonly supported queries (LHD retrieve zero results for three queries). LHD is better than SPLENDID in 13 out of 22 comparable queries. SPLENDID outperformed ANAPSID in 15 out of 24 queries while ANAPSID outperforms DARQ in 16 out of 22 commonly supported queries. For SlicedBench, FedX(cached) outperformed FedX(first run) in 29 out of 36 comparable queries. In turn FedX(first run) outperformed LHD in 17 out of 24 queries. LHD is better than SPLENDID in 17 out of 24 comparable queries. SPLENDID outperformed ANAPSID in 17 out of 26 which in turn outperformed DARQ in 12 out of 20 commonly supported queries. No results were retrieved for majority of the queries in case of ADERIS, hence not included to this section. All of the above improvements are significant based on Wilcoxon signed ranked test with significance level set to 0.05.

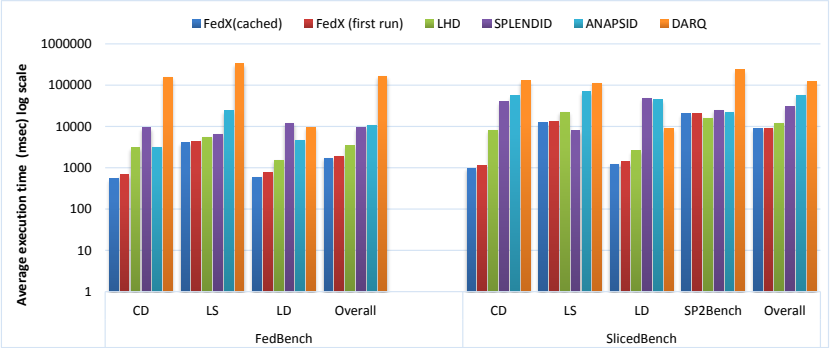


Figure 20: Overall performance evaluation (ms)

3.7 DISCUSSION

The subsequent discussion of our findings can be divided into two main categories.

3.7.1 *Effect of the source selection time*

To the best of our knowledge, the effect of the source selection runtime has not been considered in SPARQL query federation system evaluations [1; 27; 61; 73; 94] so far. However, after analysing all of the results presented above, we noticed that this metric greatly affects the overall query execution time. To show this effect, we compared the pure query execution time (excluding source selection time). To calculate the pure query execution time, we simply subtracted the source selection time from the overall query execution and plot the execution time.

We can see that the overall query execution time (including source selection) of SPLENDID is better than FedX(cached) in only one out of the 25 FedBench queries. However, our pure query execution results suggests that SPLENDID is better in 8 out of the 25 queries in terms of the pure query execution time. This means that SPLENDID is slower than FedX (cached) in 33% of the queries only due to the source selection process. Furthermore, our results also suggest that the use of SPARQL ASK queries for source selection is expensive without caching. On average, SPLENDID's source selection time is 235 ms for FedBench and 591 ms in case of SlicedBench. On the other hand, FedX (cached)'s source selection time is 8ms for both FedBench and SlicedBench. ANAPSID average source selection time for FedBench is 507 ms and 2014 ms for SlicedBench which is one of the reason of ANAPSID poor performance as compare to FedX (cached).

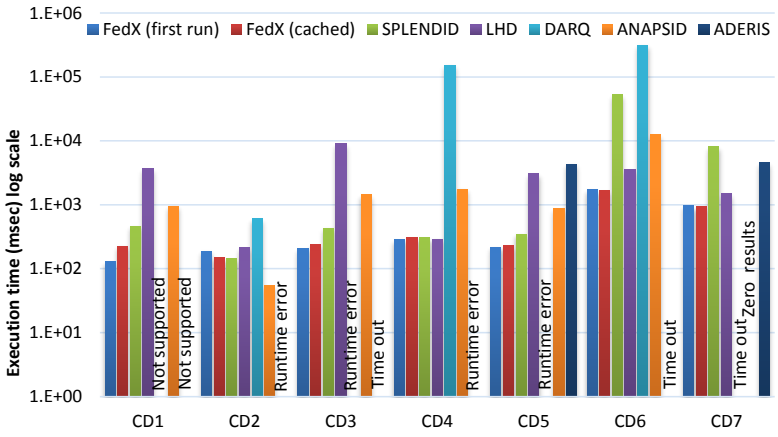


Figure 21: Comparison of pure (without source selection time) query execution time: FedBench CD queries

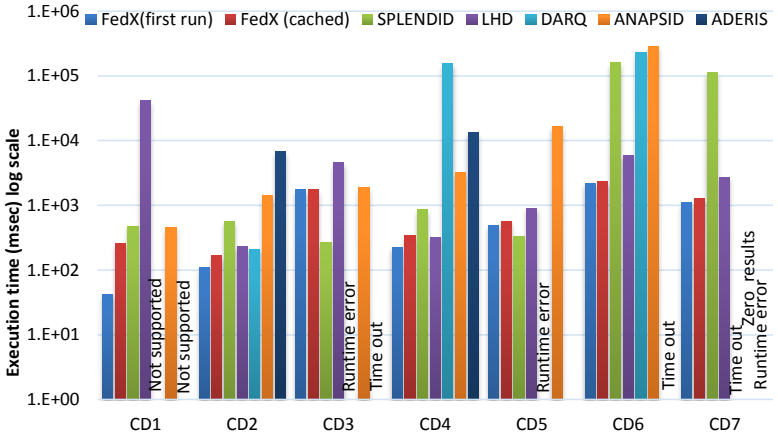


Figure 22: Comparison of pure (without source selection time) query execution time: SlicedBench CD queries

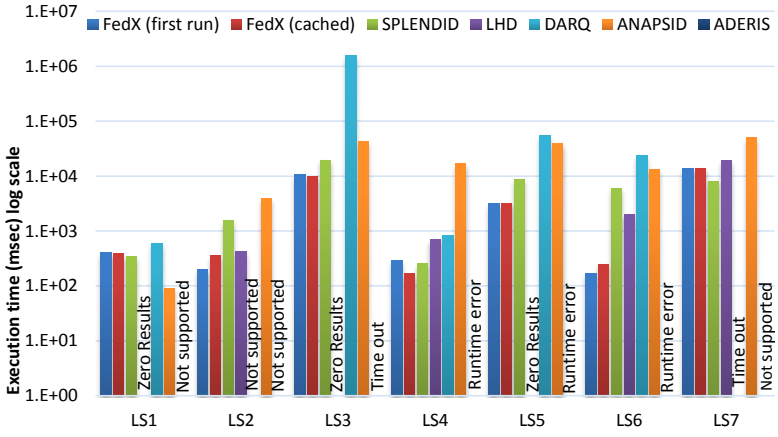


Figure 23: Comparison of pure (without source selection time) query execution time: FedBench LS queries

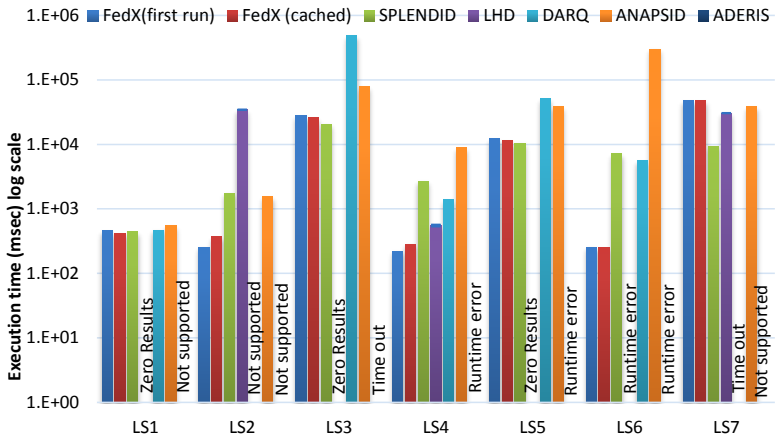


Figure 24: Comparison of pure (without source selection time) query execution time: SlicedBench LS queries

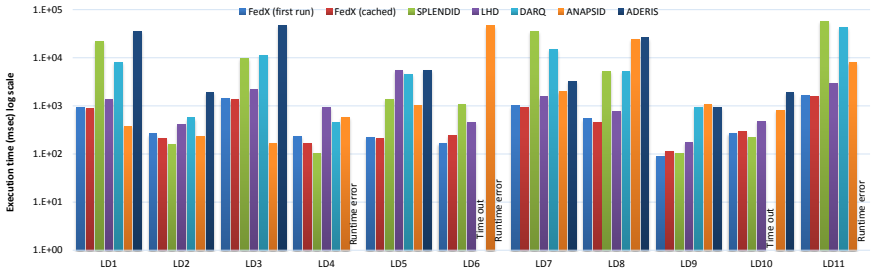


Figure 25: Comparison of pure (without source selection time) query execution time: FedBench LD queries

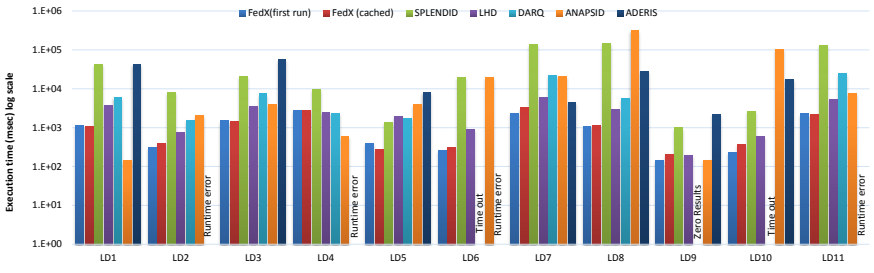
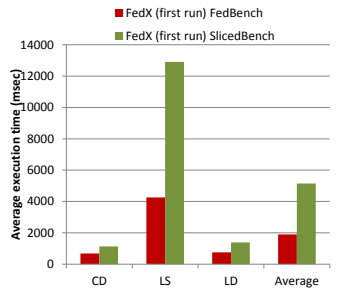


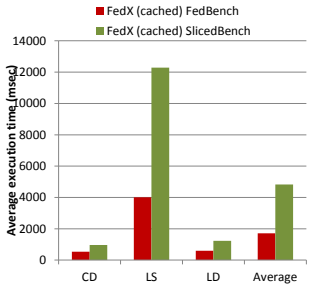
Figure 26: Comparison of pure (without source selection time) query execution time: SlicedBench LD queries

3.7.2 *Effect of the data partitioning*

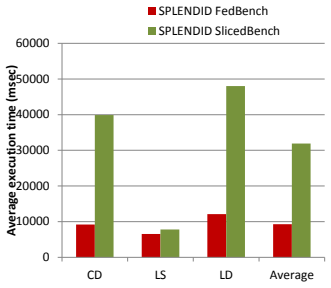
In our SlicedBench experiments, we extended FedBench to test the federation systems behaviour in highly federated data environment. This extension can also be utilized to test the capability of parallel execution of queries in SPARQL endpoint federation system. To show the effect of data partioning, we calculated the average for the query execution time of LD, CD, and LS for both the benchmarks and compared the effect on each of the selected approach. The performance of FedX(cached) and DARQ is improved with partitioning while the performance of FedX(first run), SPLENDID, ANAPSID, and LHD is reduced. As an overall evaluation result, FedX(first run)'s performance is reduced by 214%, FedX(cached)'s is reduced 199%, SPLENDID's is reduced by 227%, LHD's is reduced by 293%, ANAPSID's is reduced by 382%, and interestingly DARQ's is improved by 36%. This results suggest that FedX is the best system in terms of parallel execution of queries, followed by SPLENDID, LHD, and ANAPSID. The performance improvement for DARQ occurs due to the fact that the overflooding of endpoints with too many nested loop requests to a particular endpoint is now reduced. This reduction is due to the different distribution of the relevant results among many SPARQL endpoints. One of the reasons for the performance reduction in LHD is its significant overestimation of TP sources in SlicedBench. The reduction of both SPLENDID's and ANAPSID's performance is due to an increase in ASK operations in SlicedBench and due to the increase in triple pattern-wise selected sources which greatly affects the overall performance of the systems when no cache used.



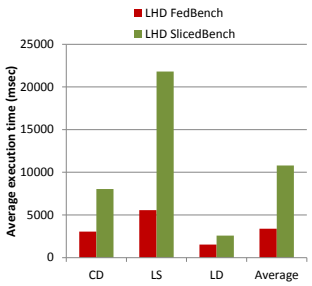
(a) FedX(first run)



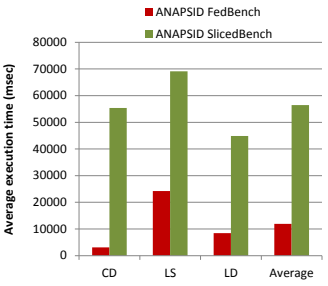
(b) FedX(cached)



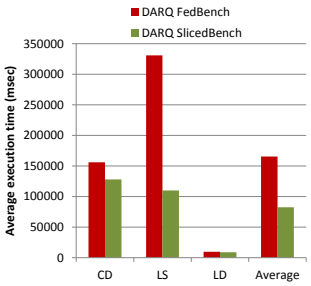
(c) SPLENDID



(d) LHD



(e) ANAPSID



(f) DARQ

Figure 27: Effect of the data partitioning

4

HYPERGRAPH-BASED SOURCE SELECTION

To ensure that a recall of 100% is achieved, most SPARQL query federation approaches [27; 57; 70; 94; 100; 77] perform *triple pattern-wise source selection* (TPWSS). In the previous chapter, it was shown that state-of-the-art SPARQL endpoint federation system greatly overestimate the set of relevant data sources. This is because most of these engine only perform TPWSS and do not consider the join between the triple patterns. Thus, it is possible that a relevant source can only answer a particular triple pattern of the query and does not *contribute* to the final result set of the complete query. This is because the results from a particular data source can be excluded after performing *joins* with the results of other triple patterns contained in the same query. We have seen that an overestimation of such sources increases the network traffic and significantly affect the overall query processing time.

We thus propose a *novel join-aware approach to TPWSS* dubbed HiBISCuS [87] and is discussed in this chapter. HiBISCuS is a labelled-hypergraph-based source selection approach which relies on a novel type of data summaries for SPARQL endpoints using the URIs authorities. Our approach goes beyond the state of the art by aiming to compute the sources that actually contribute to the final result set of an input query and that for each triple pattern. In contrast to the state of the art, HiBISCuS uses hypergraphs to detect sources that will not generate any relevant results both at triple-pattern level and at query level. By these means, HiBISCuS can generate better approximations of the sources that should be queried to return complete results for a given query. We will carry on with the motivating example given in Chapter 1 and show that HiBISCuS is able to prune one more source, i.e., it selects a total of three distinct sources (instead of four) for the query given in Listing 1.

To the best of our knowledge, this join-aware approach to TPWSS has only been tackled by an extension of the ANAPSID framework presented in [61]. Yet, this extension is based on evaluating namespaces and sending ASK queries to data sources at runtime. In contrast, HiBISCuS relies on an index that stores the authorities of the resource URIs[1] contained in the data sources at hand. Our approach proves to be more time-efficient than the ANAPSID extension as shown by our evaluation in Section 4.3.

HiBISCuS addresses the problem of source selection by several innovations. Our first innovation consists of modelling SPARQL queries

1 http://tools.ietf.org/html/rfc3986

as a sets of *directed labelled hypergraphs* (DLH). Moreover, we rely on a *novel type of summaries* which exploits the fact that the resources in SPARQL endpoints are Uniform Resource Identifiers (URIs). Our *source selection algorithm is also novel* and consists of two steps. In a first step, our approach *labels the hyperedges* of the DLH representation of an input SPARQL query q with relevant data sources. In the second step, the summaries and the type of joins in q are used to *prune the edge labels*. By these means, HiBISCuS can discard sources (without losing recall) that are not pertinent to the computation of the final result set of the query. Overall, our contributions are thus as follows:

1. We present a formal framework for modelling SPARQL queries as directed labelled hypergraphs.
2. We present a novel type of data summaries for SPARQL endpoints which relies on the authority fragment of URIs.
3. We devise a pruning algorithm for edge labels that enables us to discard irrelevant sources based on the types of joins used in a query.
4. We evaluate our approach by extending three state-of-the-art federate query engines (FedX, SPLENDID and DARQ) with HiBISCuS and comparing these extensions to the original systems. In addition, we compare our most time-efficient extension with the extension of ANAPSID presented in [61]. Our results show that we can reduce the number of source selected, the source selection time as well as the overall query runtime of each of these systems.

The structure of the rest of this chapter is as follows: we first formalize the problem statement. We present our formalization of SPARQL queries as directed labelled hypergraphs (DLH). Subsequently, we present the algorithms underlying HiBISCuS. Finally, we evaluate HiBISCuS against the state-of-the-art and show that we achieve both better source selection and runtimes on the FedBench [91] SPARQL query federation benchmark.

4.1 PROBLEM STATEMENT

In the following, we present some of the concepts and notation that are used throughout this chapter. Note some of the concepts (e.g., relevant source set etc.) are already explained in Chapter 2.

The standard for querying RDF is SPARQL.[2] The result of a SPARQL query is called its *result set*. Each element of the result set of a query is a set of *variable bindings*. *Federated SPARQL queries* are defined as queries that are carried out over a set of sources $D = \{d_1, \ldots, d_n\}$.

2 http://www.w3.org/TR/rdf-sparql-query/

Given a SPARQL query q, a source $d \in D$ is said to *contribute* to q if at least one of the variable bindings belonging to an element of q's result set can be found in d.

Definition 15 (Optimal source Set) *Let R_i be the relevant source set (formally defined in Definition 6 Chapter 2) for triple pattern t_i. The optimal source set $O_i \subseteq R_i$ for a triple pattern $t_i \in TP$ contains the relevant sources $d \in R_i$ that actually contribute to computing the complete result set of the query.*

Formally, the problem of TPWSS can then be defined as follows:

Definition 16 (Problem Statement) *Given a set D of sources and a query q, find the optimal set of sources $O_i \subseteq D$ for each triple pattern tp_i of q.*

Most of the source selection approaches [27; 57; 70; 94; 100] used in SPARQL endpoint federation systems only perform TPWSS, i.e., they find the set of relevant sources R_i for individual triple patterns of a query and do not consider computing the optimal source sets O_i. In this chapter, we present an index-assisted approach for (1) the time-efficient computation of relevant source set R_i for individual triple patterns of the query and (2) the approximation of O_i out of R_i. HiBISCuS approximates O_i by determining and removing irrelevant sources from each of the R_i. We denote our approximation of O_i by RS_i. HiBISCuS relies on DLH to achieve this goal. In the following, we present our formalization of SPARQL queries as DLH. Note we already defined the representation of SPARQL queries as directed hypergraph. We extend that definition to label the hyperedges with relevant source sets. Subsequently, we show how we make use of this formalization to approximate O_i for each tp_i.

4.2 HIBISCUS

In this section we present our approach to the source selection problem in details. We begin by presenting our approach to representing BGPs[3] of a SPARQL query as DLHs. Then, we present our approach to computing *lightweight data summaries*. Finally, we explain our approach to source selection.

4.2.1 *Queries as Directed Labelled Hypergraphs*

An important intuition behind our approach is that each of the BGP in a query can be executed separately. Thus, in the following, we will mainly focus on how the execution of a single BGP can be optimized. The representation of a query as DLH is the union of the representations of its BGPs. Note that the representations of BGPs are kept

3 http://www.w3.org/TR/sparql11-query/#BasicGraphPatterns

disjoint even if they contain the same nodes to ensure that the BGPs are processed independently. The DLH representation of a BGP is formally defined as follows:

Definition 17 *Each basic graph patterns* BGP_i *of a SPARQL query can be represented as a DLH* $HG_i = (V, E, \lambda_e, \lambda_{vt})$*, where*

1. $V = V_s \cup V_p \cup V_o$ *is the set of vertices of* HG_i*,* V_s *is the set of all subjects in* HG_i*,* V_p *the set of all predicates in* HG_i *and* V_o *the set of all objects in* HG_i*;*

2. $E = \{e_1, \ldots, e_t\} \subseteq V^3$ *is a set of directed hyperedges (short: edge). Each edge* $e = (v_s, v_p, v_o)$ *emanates from the triple pattern* $<v_s, v_p, v_o>$ *in* BGP_i*. We represent these edges by connecting the head vertex* v_s *with the tail hypervertex* (v_p, v_o)*. In addition, we use* $E_{in}(v) \subseteq E$ *and* $E_{out}(v) \subseteq E$ *to denote the set of incoming and outgoing edges of a vertex* v*;*

3. $\lambda_e : E \mapsto 2^D$ *is a hyperedge-labelling function. Given a hyperedge* $e \in E$*, its edge label is a set of sources* $R_i \subseteq D$*. We use this label to the sources that should be queried to retrieve the answer set for the triple pattern represented by the hyperedge* e*;*

4. λ_{vt} *is a vertex-type-assignment function. Given an vertex* $v \in V$*, its vertex type can be 'star', 'path', 'hybrid', or 'sink' if this vertex participates in at least one join. A 'star' vertex has more than one outgoing edge and no incoming edge. 'path' vertex has exactly one incoming and one outgoing edge. A 'hybrid' vertex has either more than one incoming and at least one outgoing edge or more than one outgoing and at least one incoming edge. A 'sink' vertex has more than one incoming edge and no outgoing edge. A vertex that does not participate in any join is of type 'simple'.*

Figure 28 shows the DLH representation of the second BGP of the motivating example query given in Listing 1.

We can now reformulate our problem statement as follows:

Definition 18 (Problem Reformulation) *Given a query* q *represented as a set of hypergraphs* $\{HG_1, \ldots, HG_x\}$*, find the labelling of the hyperedges of each hypergraph* HG_i *that leads to an optimal source selection.*

4.2.2 *Data Summaries*

HiBISCuS relies on *capabilities* to compute data summaries. Given a source d, we define a capability as a triple $(p, SA(d,p), OA(d,p))$ which contains (1) a predicate p in d, (2) the set $SA(d,p)$ of all distinct *subject authorities* of p in d and (3) the set $OA(d,p)$ of all distinct *object authorities* of p in d. In HiBISCuS, a *data summary* for a source $d \in D$ is the set $CA(d)$ of all *capabilities* of that source. Consequently,

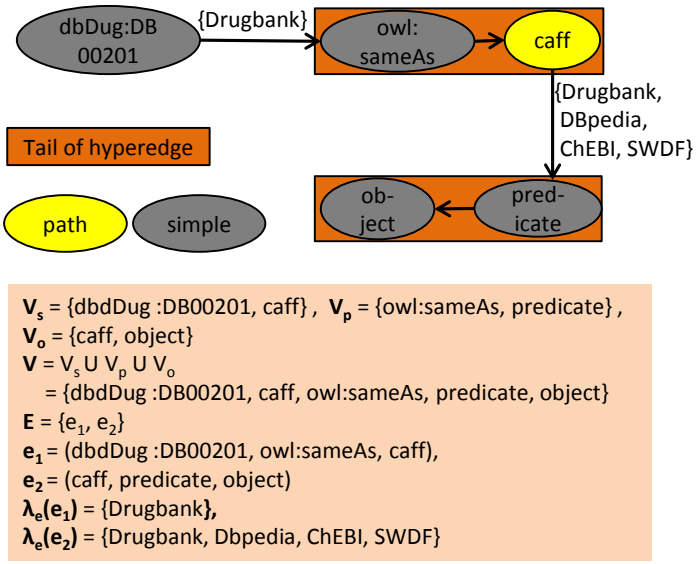


Figure 28: Labelled hypergraph of the second BGP of the motivating example query given in Listing 1.

the total number of capabilities of a source is equal to the number of distinct predicates in it.

The predicate `rdf:type` is given a special treatment: Instead of storing the set of all distinct object authorities for a capability having this predicate, we store the *set of all distinct class URIs* in d, i.e., the set of all resources that match $?x$ in the query $?y$ `rdf:type` $?x$. The reason behind this choice is that the set of distinct *classes* used in a source d is usually a small fraction of the set of all resources in d. Moreover, triple patterns with predicate `rdf:type` are commonly used in SPARQL queries. Thus, by storing the complete class URI instead of the object authorities, we might perform more accurate source selection. Listing 5 shows an example of a data summary. In the next section, we will make use of these data summaries to optimize the *TPWSS*.

4.2.3 *Source Selection Algorithm*

Our source selection comprise two steps: given a query q, we first *label all hyperedges* in each of the hypergraphs which results from the BGPs of q, i.e., we compute $\lambda_e(e_i)$ for each $e_i \in E_i$ in all $HG_i \in DHG$. We present two variations of this step and compare them in the evaluation section. In a second step, we *prune the labels of the hyperedges* assigned in the first step and compute $RS_i \subseteq R_i$ for each e_i. The pseudo-code of our approaches is shown in Algorithms 1, 2 (labelling) as well as 3 (pruning).

```
[] a ds:Service ;
   ds:endpointUrl <http://drugbank.data.source.url/sparql> ;
  ds:capability
    [ ds:predicate  owl:sameAs ;
      ds:sbjPrefixes  <http://www4.wiwiss.fu-berlin.de> ;
      ds:objAuthority  <http://dbpedia.org>, <http://bio2rdf.org> ;
      ] ;
      .
 [] a ds:Service ;
   ds:endpointUrl <http://dbpedia.data.source.url/sparql> ;
  ds:capability
    [ ds:predicate  foaf:name ;
      ds:sbjPrefixes  <http://dbpedia.org> ;
      #No objPrefixes as the object value for foaf:name is string
      ] ;
      .
 [] a ds:Service ;
   ds:endpointUrl <http://chebi.data.source.url/sparql> ;
  ds:capability
    [ ds:predicate  chebi:Status ;
      ds:sbjPrefixes   <http://bio2rdf.org> ;
      ds:objPrefixes   <http://bio2rdf.org> ;
      ] ;
  ds:capability
    [ ds:predicate  rdf:type ;
      ds:sbjPrefixes   <http://bio2rdf.org> ;
      ds:objPrefixes   <http://bio2rdf.org/chebi:Compound> ;
      #Store complete classes for rdf:type
      ] ;
        .
 [] a ds:Service ;
   ds:endpointUrl <http://data.semanticweb.org/sparql> ;
  ds:capability
    [ ds:predicate  foaf:name ;
      ds:sbjPrefixes  <http://data.semanticweb.org>  ;
       #No objPrefixes as the object value for foaf:name is string
      ] ;
      .
```

Listing 5: HiBISCuS data summaries for motivating example datasets (ref. Chapter 1). Prefixes are ignored for simplicity

4.2.3.1 *Labelling approaches*

We devised two versions of our approach to the hyperedge labelling problem, i.e., an ASK-dominant and an index-dominant version. Both take the set of all sources D, the set of all disjunctive hypergraphs DHG of the input query q and the data summaries HiBISCuS$_D$ of all sources in D as input (see Algorithms 1,2). They return a set of *labelled* disjunctive hypergraphs as output. For each hypergraph and each hyperedge, the subject, predicate, object, subject authority, and object authority are collected (Lines 2-5 of Algorithms 1,2). Edges with unbound subject, predicate, and object vertices (e.g e = (?s, ?p, ?o)) are labelled with the set of all possible sources D (Lines 6-7 of Algorithms 1,2). A data summary lookup is performed for edges with the predicate vertex `rdf:type` that have a bound object vertex. All sources with matching capabilities are selected as label of the hyperedge (Lines 9-10 of Algorithms 1,2).

The *ASK-dominant version* of our approach (see Algorithm 1, Line 11) makes use of the notion of *common predicates*. A common predicate is a predicate that is used in a number of sources above a specific threshold value θ specified by the user. A predicate is then considered a common predicate if it occurs in at least θ|D| sources. We make use of the ASK queries for triple patterns with common predicates. Here, an ASK query is sent to all of the available sources to check whether they contain the common predicate cp. Those sources which return `true` are selected as elements of the set of sources used to label that triple pattern. The results of the ASK operations are stored in a cache. Therefore, every time we perform a cache lookup before SPARQL ASK operations (Lines 14-18). In contrast, in the *index-dominant* version of our algorithm, an index lookup is performed if any of the subject or predicate is bound in a triple pattern.We will see later that the index-dominant approach requires less ASK queries than the ASK-dominant algorithm. However, this can lead to an overestimation of the set of relevant sources (see section 4.3.2).

4.2.4 *Pruning approach*

The intuition behind our pruning approach is that knowing which authorities are relevant to answer a query can be used to discard triple pattern-wise (TPW) selected sources that will not contribute to the final result set of the query. Our source pruning algorithm (ref. Algorithm 3) takes the set of all labelled disjunctive hypergraphs as input and prune labels of all hyperedges which either incoming or outgoing edges of a 'star','hybrid', 'path', or 'sink' node. Note that our approach deals with each BGP of the query separately (Line 1 of Algorithm 3).

For each node v of a DLH that is not of type 'simple', we first retrieve the sets (1) SAuth of the subject authorities contained in the

Algorithm 1 ASK-dominant hybrid algorithm for labelling all hyperedges of each disjunctive hypergraph of a SPARQL query

Require: D= $\{d_1, \ldots, d_n\}$; DHG = $\{HG_1, \ldots, HG_x\}$; $HiBISCuS_D$ //sources, disjunctive hypergraphs of a query, HiBISCuSmaries of sources

1: **for** each $HG_i \in DHG$ **do**
2: E = hyperedges (HG_i)
3: **for** each $e_i \in E$ **do**
4: s = subjvertex(e_i); p = predvertex(e_i); o = objvertex(e_i);
5: sa = subjauth(s); oa = objauth(o); //can be null i.e. for unbound s, o
6: **if** !bound(s) $\wedge$!bound(p) $\wedge$!bound(o) **then**
7: $\lambda_e(e_i)$ = D
8: **else if** bound(p) **then**
9: **if** $p = rdf : type \wedge$ bound(o) **then**
10: $\lambda_e(e_i) = HiBISCuS_D$ lookup(p, o)
11: **else if** !commonpredicate(p) $\vee$ (!bound(s) $\wedge$!bound(o)) **then**
12: $\lambda_e(e_i) = HiBISCuS_D$ lookup(sa, p, oa)
13: **else**
14: **if** cachehit(s, p, o) **then**
15: $\lambda_e(e_i)$ = cachelookup(s, p, o)
16: **else**
17: $\lambda_e(e_i)$ = ASK(s, p, o, D)
18: **end if**
19: **end if**
20: **else**
21: **Repeat** Lines 14-18
22: **end if**
23: **end for**
24: **end for**
25: **return** DHG //Set of labelled disjunctive hypergraphs

Algorithm 2 Index-dominant hybrid algorithm for labelling all hyperedges of each disjunctive hypergraph of a SPARQL query

Require: D= $\{d_1, \ldots, d_n\}$; DHG = $\{HG_1, \ldots, HG_x\}$; $HiBISCuS_D$ //sources, disjunctive hypergraphs of a query, HiBISCuSmaries of sources

1: **for** each $HG_i \in DHG$ **do**
2: E = hyperedges (HG_i)
3: **for** each $e_i \in E$ **do**
4: s = subjvertex(e_i); p = predvertex(e_i); o = objvertex(e_i);
5: sa = subjauth(s); oa = objauth(o); //can be null i.e. for unbound s, o
6: **if** !bound(s) $\wedge$!bound(p) $\wedge$!bound(o) **then**
7: $\lambda_e(e_i)$ = D
8: **else if** bound(s) $\vee$ bound(p) **then**
9: **if** bound(p) $\wedge$ $p = rdf : type \wedge$ bound(o) **then**
10: $\lambda_e(e_i) = HiBISCuS_D$ lookup(p, o)
11: **else**
12: $\lambda_e(e_i) = HiBISCuS_D$ lookup(sa, p, oa)
13: **end if**
14: **else**
15: **if** cachehit(s, p, o) **then**
16: $\lambda_e(e_i)$ = cachelookup(s, p, o)
17: **else**
18: $\lambda_e(e_i)$ = ASK(s, p, o, D)
19: **end if**
20: **end if**
21: **end for**
22: **end for**
23: **return** DHG //Set of labelled disjunctive hypergraphs

Algorithm 3 Hyperedge label pruning algorithm for removing irrelevant sources

```
Require: DHG //disjunctive hypergraphs
 1: for each HG_i ∈ DHG do
 2:   for each v ∈ vertices(HG_i) do
 3:     if λ_vt(v) ≠ 'simple' then
 4:       SAuth = ∅; OAuth = ∅;
 5:       for each e ∈ E_out(v) do
 6:         SAuth = SAuth ∪ {subjectauthories(e)}
 7:       end for
 8:       for each e ∈ E_in(v) do
 9:         OAuth = OAuth ∪ {objectauthories(e)}
10:       end for
11:       A = SAuth ∪ OAuth // set of all authorities
12:       I = A.get(1) //get first element of authorities
13:       for each a ∈ A do
14:         I = I ∩ a //intersection of all elements of A
15:       end for
16:       for each e ∈ E_in(v) ∪ E_out(v) do
17:         label = ∅ //variable for final label of e
18:         for d_i ∈ λ_e(e) do
19:           if authorities(d_i) ∩ I ≠ ∅ then
20:             label = label ∪ d_i
21:           end if
22:         end for
23:         λ_e(e) = label
24:       end for
25:     end if
26:   end for
27: end for
```

elements of the label of each outgoing edge of v (Lines 5-7 of Algorithm 3) and (2) $OAuth$ of the object authorities contained in the elements of the label of each ingoing edge of v (Lines 8-10 of Algorithm 3). Note that these are sets of sets of authorities.

`DrugBank` as single relevant source for the second triple pattern of the query given in Listing 1. The set of distinct *object* authorities (`DBpedia.org`, `bio2rdf.org` in our case) for predicate `owl:sameAs` of `DrugBank` would be retrieved from the HiBISCuS index. The set of distinct *subject* authorities (`DBpedia.org`, `bio2rdf.org` in our case) would be retrieved for all predicates (as the predicate is variable) and for all data sources (as all are relevant). Finally, the intersection of the set of authorities would again result in the same authorities set. Thus, HiBISCuS would not prune any source

For the node ?caf of query given in Listing 1 and the corresponding DLH representation given in Figure 28, we get $SAuth = \{\{$`dbpedia.org`$\}, \{$`bio2rdf.org`$\}, \{$`data.semanticweb.org`$\}, \{$`wiwiss.fu-berlin.de`$\}, \}$ for the outgoing edge. This is because the predicate is variable and and all sources are relevant. We get $OAuth = \{$`dbpedia.org, bio2rdf.org`$\}$ for the incoming edges. Now we merge these two sets to the set A of all authorities. For node ?caf, $A = \{\{$`dbpedia.org, bio2rdf.org`$\}, \{$`data.semanticweb.org`, dbpedia.org, bio2rdf.org, wiwiss.fu−berlin.de$\}$. The intersection $I = \left(\bigcap_{a_i \in A} a_i \right)$ of these elements sets is then computed. In our example, this results in $I = \{$`dbpedia.org, bio2rdf.org`$\}$. Finally, we recompute the label of each hyperedge e that is connected

to v. To this end, we compute the subset of the previous label of e which is such that the set of authorities of each of its elements is not disjoint with I (see Lines 16-23 of Algorithm 3). These are the only sources that will really contribute to the final result set of the query. In our example, `DrugBank` will be selected for first triple pattern of the BGP given in Figure 28. HiBISCuS will prune `DrugBank` and `SWDF` for the second triple pattern; the authority {`wiwiss.fu-berlin.de`} from `DrugBank` and {`data.semanticweb.org` from `SWDF` are disjoint with I. Thus, HiBISCuS selects three distinct sources (i.e., `DrugBank` for first two triple pattern and `DBpedia`, `ChEBI` for the last triple pattern) of the query given in Listing 1. Even though HiBISCuS is able to prune one more source (i.e., selected three distinct sources instead of four), still it overestimates `ChEBI` whose results will be excluded after performing the join between the last two triple patterns of the query given in Listing 1. We will solve this problem in the next chapter by using TBSS approach.

We are sure not to lose any recall by this operation because joins act in a conjunctive manner. Consequently, if the results of a data source d_i used to label a hyperedge cannot be joined to the results of at least one source of each of the other hyperedges, it is guaranteed that d_i will not contribute to the final result set of the query. In our example, this leads to d_1 being discarded from the label of the ingoing edge, while d_3 is discarded from the label of one outgoing hyperedge of node ?v1 as shown in Figure This step concludes our source selection.

4.3 EVALUATION

In this section we describe the experimental evaluation of our approach. We first describe our experimental setup in detail. Then, we present our evaluation results. All data used in this evaluation is either publicly available or can be found at the project web page.[4]

4.3.1 *Experimental Setup*

Benchmarking Environment: We used FedBench [91] for our evaluation. It is the only (to the best of our knowledge) benchmark that encompasses real-world datasets and commonly used queries within a distributed data environment. Furthermore, it is commonly used in the evaluation of SPARQL query federation systems [94; 27; 61; 77]. Each of FedBench's nine datasets was loaded into a separate physical virtuoso server. The exact specifications of the servers can be found on the project website. All experiments were ran on a machine with a 2.70GHz i5 processor, 8 GB RAM and 300 GB hard disk. The experiments were carried out in a local network, so the network costs were negligible. Each query was executed 10 times and results were aver-

4 https://code.google.com/p/hibiscusfederation/

Table 14: Comparison of index construction time and compression ratio. QTree's compression ratio is taken from [34]. (NA = Not Applicable).

	FedX	SPLENDID	LHD	DARQ	ANAPSID	Qtree	HiBISCuS
Index Generation Time (min)	NA	75	75	102	6	-	36
Compression Ratio (%)	NA	99.998	99.998	99.997	99.999	96	99.997

aged. The query timeout was set to 30min (1800s). The threshold for the ASK-dominant approach was best selected to 0.33 after analysing results of different threshold values.

Federated Query Engines: We extended three SPARQL endpoint federation engines with HiBISCuS: DARQ [70] (index-only), FedX [94] (index-free), and SPLENDID [27] (hybrid). In each of the extensions, we only replaced the source selection with HiBISCuS. The query execution mechanisms remained unchanged. We compared our best extension (i.e., SPLENDID+HiBISCuS) with ANAPSID as this engine showed competitive results w.r.t. its index compression and number of TPW sources selected.

Metrics: We compared the three engines against their HiBISCuS extension. For each query we measured (1) the total number of TPW sources selected, (2) the total number of SPARQL ASK requests submitted during the source selection, (3) the average source selection time and (4) the average query execution time. We also compare the source index/data summaries generation time and index compression ratio of various state-of-the art source selection approaches.

4.3.2 *Experimental Results*

4.3.2.1 *Index Construction Time and Compression Ratio*

Table 14 shows a comparison of the index/data summaries construction time and the compression ratio[5] of various state-of-the art approaches. A high compression ratio is essential for fast index lookup during source selection. HiBISCuS has an index size of 458KB for the complete FedBench data dump (19.7 GB), leading to a high compression ratio of 99.99%. The other approaches achieve similar compression ratios. HiBISCuS's index construction time is second only to ANAPSID's. This is due to ANAPSID storing only the distinct predicates in its index. Our results yet suggest that our index containing more information is beneficial to the query execution time on FedBench.

4.3.2.2 *Efficient Source Selection*

We define efficient source selection in terms of three metrics: (1) the total number of TPW sources selected, (2) total number of SPARQL

5 The compression ratio is given by (1 - index size/total data dump size).

Table 15: Comparison of the source selection in terms of total TPW sources selected **#T**, total number of SPARQL ASK requests **#A**, and source selection time **ST** in msec. **ST*** represents the source selection time for FedX(100% cached i.e. #A =0 for all queries) which is very rare in practical. **ST**** represents the source selection time for HiBISCuS (AD,warm) with #A =0 for all queries. (**AD** = ASK-dominant, **ID** = index-dominant, **ZR** = Zero results, **NS** = Not supported, **T/A** = Total/Avg., where Total is for #T, #A, and Avg. is ST, ST*, and ST**)

	FedX				SPLENDID			DARQ			ANAPSID			HiBISCuS(AD)				HiBISCuS(ID)		
Qry	#T	#A	ST	ST*	#T	#A	ST	#T	#A	ST	#T	#A	ST	#T	#A	ST	ST**	#T	#A	ST
CD1	11	27	285	**6**	11	26	392	NS	NS	NS	**3**	20	667	4	18	215	36	12	**0**	363
CD2	**3**	27	200	6	**3**	9	294	10	**0**	6	**3**	1	42	**3**	9	4	**3**	**3**	**0**	57
CD3	12	45	367	**8**	12	2	304	20	**0**	12	**5**	2	73	**5**	**0**	77	41	**5**	**0**	91
CD4	19	45	359	**8**	19	2	310	20	**0**	12	**5**	3	128	**5**	**0**	54	52	**5**	**0**	179
CD5	11	36	374	7	11	1	313	11	**0**	**4**	**4**	1	66	**4**	**0**	25	23	**4**	**0**	58
CD6	9	36	316	**8**	9	2	298	10	**0**	11	10	11	140	**8**	**0**	36	23	**8**	**0**	54
CD7	13	36	324	9	13	2	335	13	**0**	**6**	**6**	5	ZR	**6**	**0**	30	35	**6**	**0**	55
LS1	**1**	18	248	9	**1**	**0**	217	**1**	**0**	**4**	**1**	**0**	35	**1**	**0**	5	6	**1**	**0**	9
LS2	11	27	264	**8**	11	26	390	NS	NS	NS	12	30	548	**7**	18	118	60	**7**	**0**	118
LS3	12	45	413	**8**	12	1	310	20	**0**	9	**5**	13	808	**5**	**0**	31	27	**5**	**0**	200
LS4	**7**	63	445	**7**	**7**	2	287	15	**0**	15	**7**	1	314	**7**	**0**	8	9	**7**	**0**	15
LS5	10	54	440	**8**	10	1	308	18	**0**	13	**7**	4	885	8	**0**	20	21	8	**0**	44
LS6	9	45	430	8	9	2	347	17	**0**	**7**	**5**	13	559	7	**0**	23	22	7	**0**	42
LS7	**6**	45	389	8	**6**	1	292	6	**0**	**5**	7	2	193	**6**	**0**	18	17	**6**	**0**	24
LD1	8	27	297	8	8	1	295	11	**0**	**7**	**3**	1	428	**3**	**0**	24	19	**3**	**0**	21
LD2	**3**	27	320	**7**	**3**	1	268	3	**0**	9	**3**	**0**	34	**3**	**0**	3	5	**3**	**0**	6
LD3	16	36	330	**9**	16	1	324	16	**0**	11	**4**	2	130	**4**	**0**	31	29	**4**	**0**	48
LD4	**5**	45	326	7	**5**	2	290	5	**0**	17	**5**	**0**	33	**5**	**0**	**6**	7	**5**	**0**	10
LD5	5	27	280	8	5	2	236	13	**0**	**4**	**3**	2	210	**3**	**0**	9	9	**3**	**0**	19
LD6	14	45	385	**8**	14	1	331	14	**0**	**8**	14	12	589	**7**	**0**	32	30	**7**	**0**	136
LD7	3	18	258	7	3	2	235	4	**0**	**4**	**2**	4	223	4	**0**	7	7	4	**0**	11
LD8	15	45	337	8	15	1	333	15	**0**	**7**	9	7	1226	**5**	**0**	23	25	**5**	**0**	41
LD9	**3**	27	228	12	3	5	188	6	**0**	**3**	**3**	3	1052	**3**	9	50	3	**3**	**0**	17
LD10	10	27	274	8	10	2	309	11	**0**	**6**	**3**	4	2010	**3**	**0**	19	18	**3**	**0**	27
LD11	15	45	351	**7**	15	1	260	15	**0**	9	**5**	2	2904	7	**0**	23	24	7	**0**	42
T/A	**231**	**918**	**330**	**8**	**231**	**96**	**299**	**274**	**0**	**8**	**134**	**143**	**554**	**123**	**54**	**36**	**22**	**131**	**0**	**67**

ASK requests used to obtain (1), and (3) the TPW source selection time. Table 15 shows a comparison of the source selection approaches of FedX, SPLENDID, ANAPSID and HiBISCuS based on these three metrics. Note that FedX (100% cached) means that we gave FedX enough memory to use only its cache to perform the complete source selection. This is the best-case scenario for FedX. Overall, HiBISCuS (ASK-dominant) is the most efficient approach in terms of total TPW sources selected, HiBISCuS (Index-dominant) is the most efficient *hybrid* approach in terms of total number of ASK request used, and FedX (100% cached) is most efficient in terms of source selection time. However, FedX (100% cached) clearly overestimates the set of sources that actually contributes to the final result set of query. In the next section, we will see that this overestimation of sources greatly leads to a slightly higher overall query runtime. For ANAPSID, the results are based on Star-Shaped Group Multiple endpoint selection (SSGM) heuristics presented in its extension [61]. Further, the source selection time represents the query decomposition time as both of these steps are intermingled.

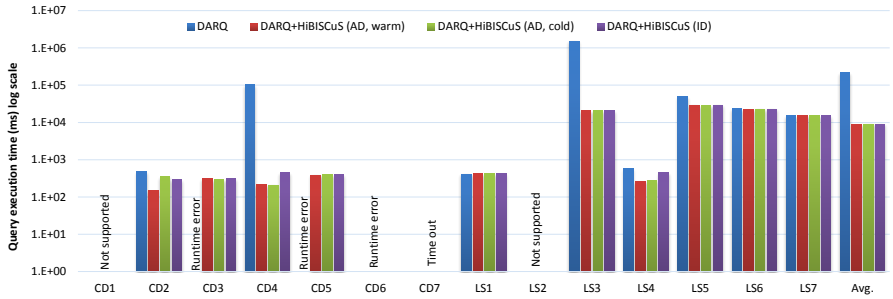


Figure 29: Query runtime of DARQ and its HiBISCuS extensions on CD and LS queries of FedBench. CD1, LS2 not supported, CD6 runtime error, CD7 time out for both. CD3 runtime error for DARQ.

4.3.2.3 *Query execution time*

The most important criterion when optimizing federated query execution engines is the query execution time. Figures 29-34 show the results of our query execution time experiments. Our main results can be summarized as follows:

(1) Overall, *the ASK-dominant (AD) version of our approach performs best*. AD is on average (over all 25 queries and 3 extensions) 27.82% faster than the index-dominant (ID) version. The reason for this improvement is due to ID overestimating sources in some queries. For example, in CD1, AD selects the optimal number of sources (i.e., 4) while ID selects 12 sources. In some cases, the overestimation of sources by ID also slows down the source pruning (e.g. CD2),

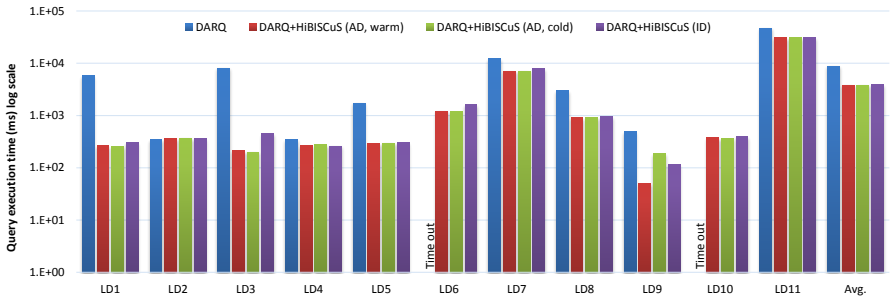


Figure 30: Query runtime of DARQ and its HiBISCuS extensions on LD queries of FedBench. LD6, LD10 timeout for DARQ.

(2) A comparison of our extensions with AD shows that *all extensions are more time-efficient than the original systems*. In particular, FedX's (100% cached) runtime is improved in 20/25 queries (net query runtime improvement of 24.61%), FedX's (cold) is improved in 25/25 queries (net improvement: 53.05%), SPLENDID 's is improved in 25/25 queries (net improvement: 82.72%) and DARQ's is improved in 21/21 (2 queries are not supported, 1 query time out, and 1 query runtime error) queries (net improvement: 92.22%). Note that these values were computed only on those queries that did not time-out. Thus, the net improvement brought about by AD is actually even better than the reported values. The reason for our slight (less than 5 msec) greater runtime for 5/25 queries in FedX (100% cached) is due to FedX (100% cached) already selecting the optimal sources for these queries. Thus, the overhead due to our pruning of the already optimal list of sources affects the overall query runtime.

(3) *Our extensions allow some queries that timed out to be carried out before the time-out*. This is especially the case for our DARQ extension, where LD6 and LD10 are carried out in 1123 msec and 377 msec respectively by DARD+AD, while they did not terminate within the time-out limit of 30 minutes on the original system.

(4) Our SPLENDID (AD) extension is 98.91% faster than ANAPSID on 24 of the 25 queries. For CD7, ANAPSID returned zero results.

An interesting observation is that FedX(100%) is better than SPLENDID in 25/25 queries and 58.17% faster on average query runtime. However, our AD extension of SPLENDID is better than AD extension of FedX(100%) in 20/25 queries and 45.20% faster on average query runtime. This means that SPLENDID is better than FedX in term of pure query execution time (excluding source selection time). A deeper investigation of the runtimes of both systems shows that SPLENDID spends on average 56.10% of total query execution on source selection. Thus, our extension showcase clearly that an efficient source selection is one of key factors in the overall optimization of federated SPARQL query processing.

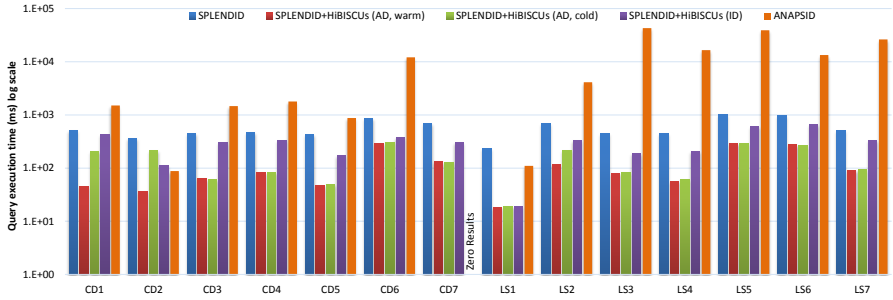


Figure 31: Query runtime of ANAPSID, SPLENDID and its HiBISCuS extensions on CD and LS queries of FedBench. We have zero results for ANAPSID CD7.

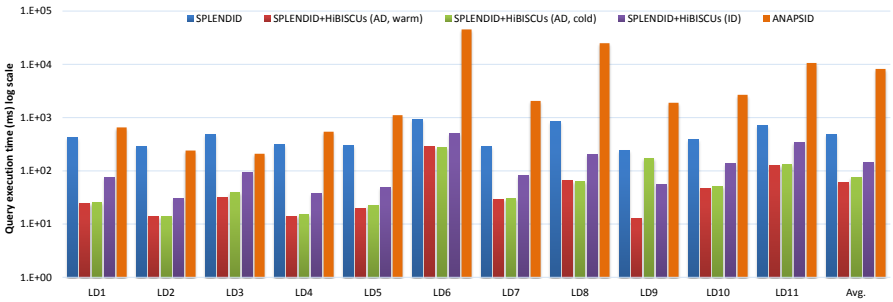


Figure 32: Query runtime of ANAPSID, SPLENDID and its HiBISCuS extensions on LD queries of FedBench. We have zero results for ANAPSID CD7.

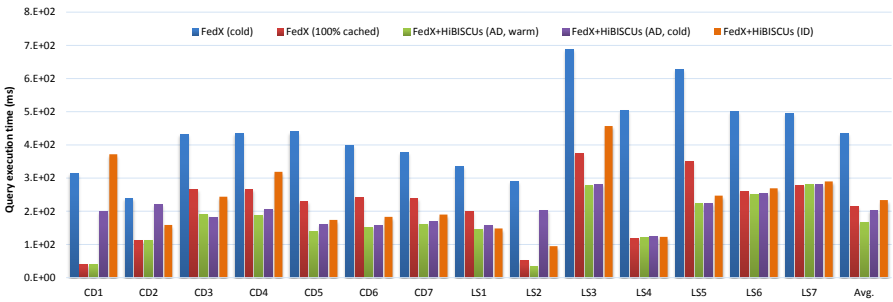


Figure 33: Query runtime of FedX and its HiBISCuS extensions on CD and LS queries of FedBench.

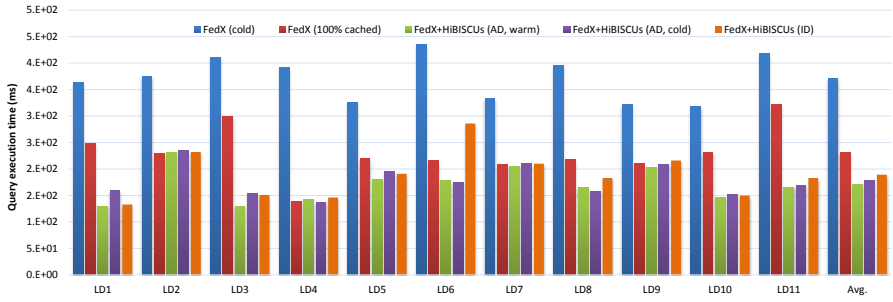


Figure 34: Query runtime of FedX and its HiBISCuS extensions on LD queries of FedBench.

5

TRIE-BASED SOURCE SELECTION

This chapter is based on [88] and provides the details of TBSS (a trie based join-aware source selection approach) and QUETSAL (a complete SPARQL query federation engine based on TBSS, HiBiSCuS and DAW). We have addressed some of limitations HiBISCuS in TBSS by using common name spaces instead of URIs authorities. We will carry on with the motivating example of Chapter 1 and show that TBSS is able to select the optimal number of sources (i.e., two) for the motivating example given in Listing 1. Recall HiBISCuS selects three distinct sources for the same query.

In previous chapter, we have discussed HIBISCuS which makes use of the distinct URIs authorities to prune irrelevant sources. We have seen that HiBISCuS can significantly remove irrelevant sources. However, it fails to prune those sources which share the same URI authority. For example, all the `Bio2RDF`[1] sources contains the same URI authority `bio2rd.org`. Similarly, all the `Linked TCGA` sources[2] [90] share the same URI authority `tcga.deri.ie`

We thus propose a *novel federated SPARQL query engine* dubbed Quetsal which combines *join-aware approach to TPWSS based on common prefixes* with *query rewriting* to outperform the state-of-the-art on federated query processing. By moving away from authorities, our approach is flexible enough to distinguish between URIs from different datasets that come from the same namespace (e.g., as in `Bio2RDF`). Moreover, our query rewriting allows reducing the number of queries shipped across the network and thus improve the overall runtime of our federated engine.

Overall, our contributions are thus as follows:

1. We present a novel source selection algorithm based on labelled hypergraphs. Our algorithm relies on a novel type of data summaries for SPARQL endpoints which relies on most common prefixes for URIs.

2. We devise a pruning algorithm for edge labels that enables us to discard irrelevant sources based on common prefixes used in joins.

3. We compared our approach with state-of-the-art federate query engines (FedX [94], SPLENDID [27], ANAPSID [1], and SPLENDID+HiBISCus [87]). Our results show that we have reduced the number of sources selected (without losing recall), the source

1 Bio2RDF: `http://download.bio2rdf.org/release/2/release.html`
2 Linked TCGA: `http://tcga.deri.ie/`

selection time as well as the overall query runtime by executing many remote joins (i.e., joins shipped to SPARQL endpoints).

The rest of the chapter is structured as follows: we first explain TBSS source selection in details. We then present the general architecture of QUETSAL and the SPARQL 1.1 query writer. Finally, we evaluate our approach against the state-of-the-art and show that we achieve both better source selection and runtimes on the FedBench [91] SPARQL query federation benchmark.

5.1 TBSS

Like HiBISCuS, TBSS relies on directed labelled hypergraphs (DLH) to achieve this goal. Thus, the problem statement remains the same, i.e., given a query q represented as a set of hypergraphs $\{HG_1, \ldots, HG_x\}$, find the labelling of the hyperedges of each hypergraph HG_i that leads to an optimal source selection. In this section we present our approach in details. First, we explain our *lightweight data summaries* construction based on common prefixes, followed by the source selection algorithms and then we explain the SPARQL query rewriting and execution.

5.1.1 *TBSS Data Summaries*

TBSS relies on *capabilities* to compute data summaries. Given a source d, we define a capability as a triple $(p, SP(d,p), OP(d,p))$ which contains (1) a predicate p in d, (2) the set $SP(d,p)$ of all distinct *subject prefixes* of p in d and (3) the set $OA(d,p)$ of all distinct *object prefixes* of p in d. In TBSS, a *data summary* for a source $d \in D$ is the set $CA(d)$ of all *capabilities* of that source. Consequently, the total number of capabilities of a source is equal to the number of distinct predicates in it. A sample TBSS data summaries for the motivating example is provided in supplementary material[3].

The innovation behind the TBSS data summaries is to use common prefixes to characterize resources in data sets. By going beyond mere authorities, we can ensure that URIs that come from different datasets published by the same authority can still be differentiated. To achieve this goal, we rely on prefix trees (short: tries). The idea here is to record the most common prefixes of URIs used as subject or object of any given property and store those in the TBSS index.

Let ρ be a set of resources for which we want to find the most common prefixes (e.g., the set of all subject resources for a given predicate). We begin by adding all the resources containing the same *URI authority* in ρ to a trie. While we use a character-by-character insertion in our implementation, we present word-by-word insertion for

3 QUETSAL supplementary material: http://goo.gl/EFNu52

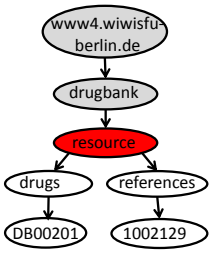


Figure 35: Trie of URIs.

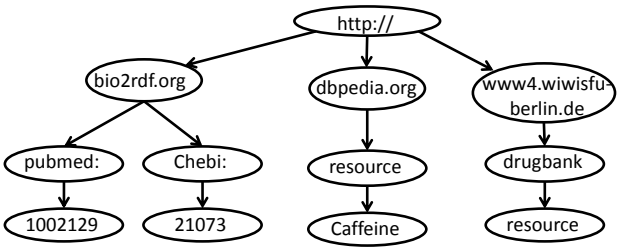


Figure 36: Trie of all Prefixes.

the sake of clarity and space in the paper. Inserting the resources `drugbank-drug:DB00201` and `drugbank-ref:1002129` from DrugBank leads to the trie shown Figure 35. We consider a node to be the end of a common prefix if (1) it is not the root of the tree and (2) the branching factor of the said node is higher than a preset threshold. For example, if our threshold were 1, the node `resource` would be the end of a common prefix. By inserting all nodes from ρ into a trie and marking all ends of common prefix, we can now compute all common prefixes for ρ by simply traversing computing all paths from the root to the marked nodes. In our example, we get exactly one common prefix for the branching limit equal to 1.

The predicate `rdf:type` is given a special treatment: Instead of storing the set of all distinct object prefixes for a capability having this predicate, we store the *set of all distinct class URIs* in d, i.e., the set of all resources that match ?x in the query ?y `rdf:type` ?x. The reason behind this choice is that the set of distinct *classes* used in a source d is usually a small fraction of the set of all resources in d. Moreover, triple patterns with predicate `rdf:type` are commonly used in SPARQL queries. Thus, by storing the complete class URI instead of the object authorities, we might perform more accurate source selection. Listing 6 shows TBSS's data summaries (branching limit = 1) for the three datasets of our motivating example. In the next section, we will make use of these data summaries to optimize the *TPWSS*.

5.1.2 *TBSS Source Selection Algorithm*

TBSS's source selection comprises two steps: Given a query q, we first *label all hyperedges* in each of the hypergraphs which results from the BGPs of q, i.e., we compute $\lambda_e(e_i)$ for each $e_i \in E_i$ in all $HG_i \in DHG$. In a second step, we *prune the labels of the hyperedges* assigned in the first step and compute $RS_i \subseteq R_i$ for each e_i. The pseudo-code of our approaches is shown in Algorithm 4 (labelling) as well as Algorithm 5 (pruning).

5.1.2.1 *Hyperedge Labelling*

```
[] a ds:Service ;
   ds:endpointUrl <http://drugbank.data.source.url/sparql> ;
  ds:capability
    [ ds:predicate  owl:sameAs ;
      ds:sbjPrefixes  <http://www4.wiwiss.fu-berlin.de/drugbank/
          resource/> ;
      ds:objAuthority  <http://DBpedia.org/resource/Caffeine>, <http
          ://bio2rdf.org/pubmed:1002129> ;
      ] ;
      .
 [] a ds:Service ;
   ds:endpointUrl <http://dbpedia.data.source.url/sparql> ;
  ds:capability
    [ ds:predicate  foaf:name ;
      ds:sbjPrefixes  <http://dbpedia.org/resource/> ;
      #No objPrefixes as the object value for foaf:name is string
      ] ;
      .
 [] a ds:Service ;
   ds:endpointUrl <http://chebi.data.source.url/sparql> ;
  ds:capability
    [ ds:predicate  chebi:Status ;
      ds:sbjPrefixes   <http://bio2rdf.org/chebi:> ;
      ds:objPrefixes   <http://bio2rdf.org/chebi:> ;
      ] ;
  ds:capability
    [ ds:predicate  rdf:type ;
      ds:sbjPrefixes   <http://bio2rdf.org/chebi:21073> ;
      ds:objPrefixes   <http://bio2rdf.org/chebi:Compound> ;
      ] ;
        .
 [] a ds:Service ;
   ds:endpointUrl <http://data.semanticweb.org/sparql> ;
  ds:capability
    [ ds:predicate  foaf:name ;
      ds:sbjPrefixes <http://data.semanticweb.org/person/steve-tjoa>;
       #No objPrefixes as the object value for foaf:name is string
      ] ;
      .
```

Listing 6: TBSS data summaries for motivating example datasets (ref. Chapter 1) with branching Limit = 1. Prefixes are ignored for simplicity

Algorithm 4 QUETSAL's hyperedge labelling algorithm

Require: D= $\{d_1, \ldots, d_n\}$; DHG = $\{HG_1, \ldots, HG_x\}$; QUETSAL_D //sources, disjunctive hypergraphs of a query, QUETSAL sumaries of sources

```
1: for each HG_i ∈ DHG do
2:     E = hyperedges (HG_i)
3:     for each e_i ∈ E do
4:         s = subjvertex(e_i); p = predvertex(e_i); o = objvertex(e_i);
5:         sa = subjauth(s); oa = objauth(o); //can be null i.e. for unbound s, o
6:         if !bound(s) ∧ !bound(p) ∧ !bound(o) then
7:             λ_e(e_i) = D
8:         else if bound(p) then
9:             if p = rdf : type ∧ bound(o) then
10:                λ_e(e_i) = QUETSAL_D lookup(p, o)
11:            else if !commonpredicate(p) ∨ (!bound(s) ∧ !bound(o)) then
12:                λ_e(e_i) = QUETSAL_D lookup(sa, p, oa)
13:            else
14:                if cachehit(s, p, o) then
15:                    λ_e(e_i) = cachelookup(s, p, o)
16:                else
17:                    λ_e(e_i) = ASK(s, p, o, D)
18:                end if
19:            end if
20:        else
21:            Repeat Lines 14-18
22:        end if
23:    end for
24: end for
25: return DHG //Set of labelled disjunctive hypergraphs
```

The hyperedge labelling Algorithm 4 takes the set of all sources D, the set of all disjunctive hypergraphs DHG (one per BGP of the query) of the input query q and the data summaries QUETSAL_D of all sources in D as input and returns a set of *labelled* disjunctive hypergraphs as output. For each hypergraph and each hyperedge, the subject, predicate, object, subject authority, and object authority are collected (Lines 2-5 of Algorithms 4). Edges with unbound subject, predicate, and object vertices (e.g e = (?s, ?p, ?o)) are labelled with the set of all possible sources D (Lines 6-7 of Algorithms 4). A data summary lookup is performed for edges with the predicate vertex `rdf:type` that have a bound object vertex. All sources with matching capabilities are selected as label of the hyperedge (Lines 9-10 of Algorithms 4).

Algorithm 4 makes use of the notion of *common predicates*. A common predicate is a predicate that is used in a number of sources above a specific threshold value θ specified by the user. A predicate is then considered a common predicate if it occurs in at least $\theta|D|$ sources. We make use of the ASK queries for triple patterns with common predicates. Here, an ASK query is sent to all of the available sources to check whether they contain the common predicate cp. Those sources which return `true` are selected as elements of the set of sources used to label that triple pattern. The results of the ASK operations are stored in a cache. Therefore, every time we perform a cache lookup before SPARQL ASK operations (Lines 14-18).

5.1.3 *TBSS Pruning approach*

The intuition behind our pruning approach is that knowing which stored prefixes are relevant to answer a query can be used to discard triple pattern-wise (TPW) selected sources that will not contribute to the final result set of the query. Our source pruning algorithm (ref. Algorithm 5) takes the set of all labelled disjunctive hypergraphs as input and prune labels of all hyperedges which either incoming or outgoing edges of a 'star','hybrid', 'path', or 'sink' node. Note that our approach deals with each BGP of the query separately (Line 1 of Algorithm 5).

For each node v of a DLH that is not of type 'simple', we first retrieve the sets (1) $SPrefix$ of the subject prefixes contained in the elements of the label of each outgoing edge of v (Line 5-7 of Algorithm 5) and (2) $OPrefix$ of the object prefixes contained in the elements of the label of each ingoing edge of v (Line 8-10 of Algorithm 5). Note that these are sets of sets of prefixes. For the node ?caff of the query in our running example given in Listing 1, we get $SPrefix = \{\{$`http://www4.wiwiss.fu-berlin.de/drugbank/resource`$\}$, $\{$`http://dbpedia.org/resource/`$\}, \{$`http://bio2rdf.org/chebi:`$\}$, $\{$`http://data.semanticweb.org/person/steve-tjoa`$\}$, $\{$`http://bio2rdf.org/chebi:21073`$\}\}$ for the outgoing edge and $OAuth = \{\{$`http://dbpedia.org/resource/Caffeine`$\}$, $\{$`http://bio2rdf.org/pubmed:1002129`$\}\}$ for the incoming edges. Now we merge these two sets to the set P of all prefixes (Line 11 of Algorithm 5). Next, we plot all of the prefixes of P in to a trie (no branching limit) shown in Figure 36. Now we check each prefix in P whether at ends at a child node of trie T or not. If a prefix does not end at child node then we get all of the paths from the prefix last node (say n) to each leaf of n. In our example, `http://dbpedia.org/resource/`, `http://bio2rdf.org/chebi:` does not end at leaf nodes (see Figure 36), so they will be replaced with `http://dbpedia.org/resource/Caffeine`, `http://bio2rdf.org/chebi:21073`, respectively in P (Line 13-21 of Algorithm 5). The intersection $I = \left(\bigcap_{p_i \in P} p_i \right)$ of these elements sets is then computed. In our example, this results in $I = \{$`http://dbpedia.org/resource/Caffeine`$\}$. Finally, we recompute the label of each hyperedge e that is connected to v. To this end, we compute the subset of the previous label of e which is such that the set of prefixes of each of its elements is not disjoint with I (see Lines 27 onwards of Algorithm 5). These are the only sources that will really contribute to the final result set of the query. In our example, `ChEBI` will be removed since the `ChEBI` subject prefixes does not intersect (i.e., disjoint) with the elements in I and `DBpedia` will be selected since its subject prefix `http://dbpedia.org/resource/` was replaced

with
`http://dbpedia.org/resource/Caffeine` which intersect with I.

Thus, TBSS selects the optimal two distinct sources (i.e., `DrugBank` for first two triple pattern and `DBpedia` for the last triple pattern) of the query given in Listing 1. Recall, HiBISCuS selected three distinct sources (i.e., `DrugBank` for first two triple pattern and `DBpedia`, `ChEBI` for the last triple pattern) for the same query.

We are sure not to lose any recall by this operation because joins act in a conjunctive manner. Consequently, if the results of a data source d_i used to label a hyperedge cannot be joined to the results of at least one source of each of the other hyperedges, it is guaranteed that d_i will not contribute to the final result set of the query.

Algorithm 5 TBSS's hyperedge label pruning algorithm for removing irrelevant sources

```
Require: DHG //disjunctive hypergraphs
 1: for each HG_i ∈ DHG do
 2:     for each v ∈ vertices(HG_i) do
 3:         if λ_vt(v) ≠ 'simple' then
 4:             SPrefix = ∅; OPrefix = ∅;
 5:             for each e ∈ E_out(v) do
 6:                 SPrefix = SPrefix ∪ {subjectPrefixes(e)}
 7:             end for
 8:             for each e ∈ E_in(v) do
 9:                 OPrefix = OPrefix ∪ {objectPrefixes(e)}
10:             end for
11:             P = SPrefix ∪ OPrefix // set of all prefixes
12:             T = getTrie(P) //get Trie of all prefixes, no branching limit
13:             for each p ∈ P do
14:                 if !isLeafPrefix(p,T) // prefix does not end at leaf node of Trie then
15:                     C = getAllChildPaths(p) // get all paths from prefix last node n to
                        each leaf of n
16:                     A = ∅ //to store all possible prefixes of a given prefix
17:                     for each c ∈ C do
18:                         A = A ∪ p.Concatenate(c)
19:                     end for
20:                     P.replace(p,A) // replace p with its all possible prefixes
21:                 end if
22:             end for
23:             I = P.get(1) //get first element of prefixes
24:             for each p ∈ P do
25:                 I = I ∩ p //intersection of all elements of A
26:             end for
27:             for each e ∈ E_in(v) ∪ E_out(v) do
28:                 label = ∅ //variable for final label of e
29:                 for d_i ∈ λ_e(e) do
30:                     if prefixes(d_i) ∩ I ≠ ∅ then
31:                         label = label ∪ d_i
32:                     end if
33:                 end for
34:                 λ_e(e) = label
35:             end for
36:         end if
37:     end for
38: end for
```

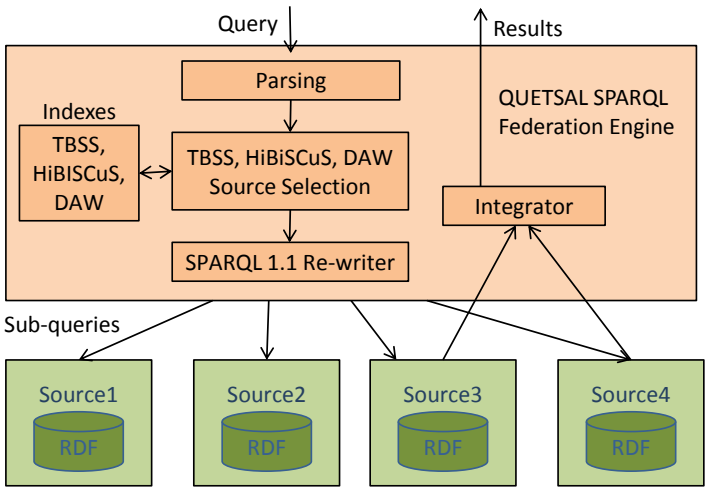


Figure 37: QUETSAL's architecture.

5.2 QUETSAL

In this section, we explain general architecture of the QUETSAL SPARQL endpoint federation engine and the corresponding SAPRQL 1.1 query rewriting.

5.2.1 QUETSAL's *Architecture*

Figure 37 shows the general architecture of QUETSAL that combines HiBISCuS, TBSS, and DAW with SPARQL 1.1 query rewriting to form a complete SPARQL endpoint federation engine. Note DAW is source selection approach for duplicate-aware federated query processing. It can be directly combined to any of the existing triple pattern-wise source selection approaches and is explained in the next chapter. Given a query, the first step is to parse the query and get the individual triple patterns. The next step is to perform triple pattern-wise source selection by using HiBISCuS or TBSS. We can combine DAW with HiBISCuS or TBSS to perform duplicate-aware source selection. Using the triple pattern-wise source selection information, QUETSAL converts the original SPARQL 1.0 query into the corresponding SPARQL 1.1 query (using the SPARQL `SERVICE` clause). The SPARQL 1.1 query rewriting is explained in subsequent section. Finally, the SPARQL 1.1 query is executed on top of Sesame and the results are integrated.

5.2.2 QUETSAL's *SPARQL 1.1 Query Re-writing*

QUETSAL converts each SPARQL 1.0 query into corresponding SPARQL 1.1 query. Before going into the details of our SPARQL 1.1 query rewriting, we first introduce the notion of exclusive groups (used in SPARQL 1.1 query rewrite) in the SPARQL query.

Exclusive Groups

In normal SPARQL query (i.e., not a federated query) execution, the user sends a complete query to the SPARQL endpoint and gets the results back from the endpoint, i.e., the complete query is executed at SPARQL endpoint. Unlike normal SPARQL query execution, in general, the federated engine sends sub-queries to the corresponding SPARQL endpoints and get the sub-queries results back which are locally integrated by using different *join* techniques. The local execution of joins results in high costs, in particular when intermediate results are large[94]. To minimize these costs, many of the existing SPARQL federation engines [94; 27; 1] make use of the notion of exclusive groups which is formally defined as:

Definition 19 *Let BGP = {$tp_1, \ldots, tp_m$} be a basic graph patterns containing a set of triple patterns $tp_1, \ldots, tp_m$, $D = \{d_1, \ldots, dn\}$ be the set of distinct data sources, and $R_{tp_i} = \{d_1, \ldots, d_o\} \subseteq D$ be the set of relevant data sources for triple pattern tp_i. We define $EG_d = \{tp_1, \ldots, tp_p\} \subseteq BGP$ be the exclusive groups of triple patterns for a data soruce $d \in D$ s.t. $\forall_{tp_i \in EG_d} R_{tp_i} = \{d\}$, i.e., the triple patterns whose single relevant source is d.*

The advantage of exclusive groups (size greater than 1) is that they can be combined together (as a conjunctive query) and send to the corresponding data source (i.e., SPARQL endpoints) in a single subquery, thus greatly minimizing: (1) the number of remote requests, (2) the number of local joins, (3) the number of irrelevant intermediate results and (4) the network traffic [94]. This is because in many cases the intermediate results of the individual triple patterns are often excluded after performing join with the intermediate results of another triple pattern in the same query. On the other hand, the triple pattern joins in the exclusive groups are directly performed by the data source itself (we call them remote joins), thus all intermediate irrelevant results are directly filtered without sending via network. Correctness is guaranteed as no other data source can contribute to the group of triple patterns with further information.

Consider the query given in Listing 1 of the supplementary material. The last three triple patterns form an exclusive group, since `DrugBank` is the single relevant source for these triple patterns. Thus these triple patterns can be directly executed by `DrugBank`.

Our SPARQL 1.0 to SPARQL 1.1 query rewrite makes use of the exclusive groups, SPARQL SERVICE, and SPARQL UNION clauses as follows: (1) Identify exclusive groups from the results of the source selection, (2) group each exclusive group of triple patterns into a separate SPARQL SERVICE, and (3) write a separate SPARQL SERVICE clause for each triple patterns (which are not part of any exclusive group) and for each relevant source of that triple pattern and union them using SPARQL UNION clause.

```
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/
    drugbank/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX purl: <http://purl.org/dc/elements/1.1/>
SELECT ?drug ?keggUrl ?chebiImage WHERE {
  ?keggDrug bio2rdf:url ?keggUrl.                    //ChEBI, KEGG
  ?drug rdf:type drugbank:drugs .                    //Drugbank
  ?drug drugbank:keggCompoundId ?keggDrug .          //Drugbank
  ?drug drugbank:genericName ?drugBankName .         //Drugbank
 }
}
```

Listing 7: QUETSAL's SPARQL 1.0 query

```
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/
    drugbank/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX purl: <http://purl.org/dc/elements/1.1/>
SELECT ?drug ?keggUrl ?chebiImage WHERE {
  { SERVICE <http://chebi.sparql.endpoint.url> {?keggDrug bio2rdf:url
      ?keggUrl.}}
 UNION
 { SERVICE <http://kegg.sparql.endpoint.url> {?keggDrug bio2rdf:url ?
     keggUrl.}}
 SERVICE  <http://drugbank.sparql.endpoint.url> {
   ?drug rdf:type drugbank:drugs .
   ?drug drugbank:keggCompoundId ?keggDrug .
   ?drug drugbank:genericName ?drugBankName .
  }
}
```

Listing 8: SPARQL 1.1 re-write of the query given in Listing 7

A SPARQL 1.1 query rewrite of the SPARQL 1.0 query given in Listing 1 of the supplementary material is shown in Listing 2 of the supplementary material. The exclusive group of triple patterns (i.e., last three triple patterns) are grouped into `DrugBank` SERVICE. Since both KEGG and `ChEBI` are the relevant data sources for the first triple pattern, a separate SPARQL SERVICE is used for each of the data source and the results are combined using SPARQL UNION clause. The final SPARQL 1.1 query is then directly executed using Sesame API. The number of remote joins for an exclusive group is one less than the number of triple patterns in that group. For example, the number of remote joins in our SPARQL 1.1 query is 2. The number of remote joins for a complete query is the sum of the number of remote joins of all exclusive groups in that query.

5.3 EVALUATION

In this section we describe the experimental evaluation of our approach. We first describe our experimental setup in detail. Then, we

present our evaluation results. All data used in this evaluation is either publicly available or can be found at the project web page.[4]

5.3.1 *Experimental Setup*

Benchmarking Environment: We used FedBench [91] for our evaluation as it is commonly used in the evaluation of SPARQL query federation systems [94; 27; 61; 77]. FedBench comprises a total of 25 queries out of which 14 queries are for SPARQL endpoint federation approaches and 11 queries for Linked Data federation approaches. We considered all of the 14 SPARQL endpoint federation queries. Each of FedBench's nine datasets was loaded into a separate physical virtuoso server. The exact specification of the servers can be found on the project website. All experiments were ran on a machine with a 2.9GHz i7 processor, 8 GB RAM and 500 GB hard disk. The experiments were carried out in a local network, so the network costs were negligible. Each query was executed 10 times and results were averaged. The query timeout was set to 10min (1800s). The threshold for the labelling Algorithm 4 was best set to 0.33 based on a prior experiments.
Federated Query Engines: We compared QUETSAL with the latest versions of FedX [94], ANAPSID [1], SPLENDID [27], and the best HiBISCuS extension (i.e., SPLENDID+HiBISCuS). To the best of our knowledge, these are the most up-to-date SPARQL endpoint federation engines.
Metrics: We compared the selected engines based on: (1) the total number of TPW sources selected, (2) the total number of SPARQL ASK requests submitted during the source selection, (3) the average source selection time, (4) the number of remote joins produced by the source selection (ref. Section 5.2.2), and (5) the average query execution time. We also compared the source index/data summaries generation time and index compression ratio of various state-of-the art source selection approaches.

5.3.2 *Experimental Results*

5.3.2.1 *Index Construction Time and Compression Ratio*

Table 16 shows a comparison of the index/data summaries construction time and the compression ratio[5] of various state-of-the art approaches. A high compression ratio is essential for fast index lookup during source selection. QUETSAL-B1 (i.e., QUETSAL with trie branching limit = 1) has an index size of only 520KB and QUETSAL-B5 (branching limit =5) has an index size of 1MB for the complete FedBench

4 QUETSAL home page: https://code.google.com/p/quetsal/
5 The compression ratio is given by (1 - index size/total data dump size).

Table 16: Comparison of Index Generation Time **IGT** in minutes, Compression Ratio **CR**, Average (over all 14 queries) Source Selection Time **ASST**, and over all overestimation of relevant data sources **OE** . QTree's compression ratio is taken from [34] and DARQ, LHD results are calculated from [83]. (NA = Not Applicable, B1 = QUETSAL branching limit 1, B2 = QUESTSAL branching limit 2 and so on).

	FedX	SPLENDID	LHD	DARQ	ANAPSID	Qtree	HiBISCuS	QUETSAL				
								B1	B2	B3	B4	B5
IGT	NA	110	100	115	5	-	36	40	45	51	54	63
CR	NA	99.998	99.998	99.997	99.999	96.000	99.997	99.997	99.997	99.996	99.995	99.993
ASST	5.5	332	14	8	136	-	26	101	108	116	129	160
OE	50%	50%	69%	62%	16.2%	-	11.80%	9.40%	9.40%	9.40%	4.20%	4.20%

data dump (19.7 GB), leading to a high compression ratio of 99.99% in all B1-B5. The other approaches achieve similar compression ratios except Qtree. Our index construction time ranges from 40min to 63min for QUETSAL-B1 to QUETSAL-B5. This is the first index construction time, later updates are possible per individual capability (as they are independent of each others) of the data source by using simple SPARQL update operations (our index is in RDF). QUETSAL-B4 is the best choice in terms of source selection accuracy and average execution time (shown in next section).

5.3.2.2 *Efficient Source Selection*

We define efficient source selection in terms of three metrics: (1) the total number of TPW sources selected, (2) total number of SPARQL ASK requests used to obtain (1), (3) the TPW source selection time, (4) and the number of remote joins (ref. Section 5.2.2) produced by the source selection. Table 17 shows a comparison of the source selection of the selected engines. Note that FedX (100% cached) means that the complete source selection is done by only using cache, i.e., no SPARQL ASK request is used. This is the best-case scenario for FedX and rare in practical applications. Overall, QUETSAL-B4 is the most efficient approach in terms of total TPW sources selected, the total number of ASK request used, and the total number of remote joins (equal with ANAPSID) produced. FedX (100% cached) is most efficient in terms of source selection time. However, FedX (100% cached) clearly overestimates the set of sources that actually contributes to the final result set of query.

Table 17: Comparison of the source selection in terms of total TPW sources selected **#T**, total number of SPARQL ASK requests **#A**, source selection time **ST** in msec, and total number of remote joins **#J** generated by the source selection. **ST*** represents the source selection time for FedX (100% cached i.e. #A =0 for all queries), #T1, #T4 QUETSAL total TPW sources selected for branching limit 1 and 4, respectively, #J1, #J4 QUETSAL number of remote joins for branching limit 1 and 4, respectively. (**T/A** = Total/Avg., where Total is for #T, #A, and Avg. is ST, ST*)

	FedX					SPLENDID				ANAPSID				HiBISCuS				QUETSAL							O ptimal	
Qry	#T	#A	ST	ST*	#J	#T	#A	ST	#J	#T	#A	ST	#J	#T	#A	ST	#J	#T1	#T4	#A	ST1	ST4	#J1	#J4	#T	#J
CD1	11	27	221	5	0	11	26	293	0	3	19	227	1	4	18	36	0	4	4	18	50	139	0	0	3	1
CD2	3	27	211	6	1	3	9	293	1	3	1	46	1	3	9	8	1	3	3	9	19	26	1	1	3	1
CD3	12	45	255	5	2	12	2	400	2	5	2	82	3	5	0	45	3	5	5	0	76	122	3	3	5	3
CD4	19	45	265	7	1	19	2	340	1	5	3	74	3	5	0	52	3	5	5	0	207	210	3	3	5	3
CD5	11	36	250	8	1	11	1	330	1	4	1	54	2	4	0	23	2	4	4	0	124	160	2	2	4	2
CD6	9	36	245	5	0	9	2	303	0	9	10	35	0	8	0	23	0	8	8	0	85	90	0	0	8	2
CD7	13	36	252	6	0	13	2	354	0	6	5	32	1	6	0	34	1	6	6	0	114	162	1	1	6	3
LS1	1	18	200	5	0	1	0	189	0	1	0	55	0	1	0	9	0	1	1	0	12	11	0	0	1	0
LS2	11	27	230	6	0	11	26	592	0	15	19	356	0	7	18	28	0	5	4	18	226	270	0	0	3	0
LS3	12	45	267	5	2	12	1	334	2	5	11	262	2	5	0	27	3	5	5	0	228	250	3	3	5	3
LS4	7	63	288	7	5	7	2	299	5	7	0	333	5	7	0	9	5	7	7	0	22	26	5	5	7	5
LS5	10	54	243	5	2	10	1	355	2	7	4	105	3	8	0	26	2	8	7	0	88	140	2	3	6	3
LS6	9	45	264	3	1	9	2	262	1	5	12	180	2	7	0	22	1	7	5	0	98	133	1	3	5	3
LS7	6	45	254	5	1	6	1	252	1	5	2	81	2	6	0	23	1	6	6	0	71	80	1	1	6	2
T/A	134	549	246	**5**	16	134	77	328	16	80	89	137	**25**	76	**45**	26	22	74	**70**	**45**	101	129	22	**25**	67	31

5.3.2.3 *Query execution time*

As mentioned before we considered all of the 14 FedBench's SPARQL endpoint federation queries. However, QUETSAL gives runtime error for three queries (reason explained at the end of this section). Thus, the selected federation engines were compared for the remaining 11 queries shown in Figure 38. Overall, QUETSAL-B1 performs best in terms of the number of queries count for which its runtime is better than others. QUETSAL-B1 is better than FedX in 6/11 queries. FedX is better than ANAPSID in 7/11 which in turn better than QUETSAL-B4 in 7/11 queries. QUETSAL-B4 is better than SPLENDID+HiBISCuS in 6/11 queries which in turn better than SPLENDID in 10/11 queries. However, QUETSAL-B4 is better of all in terms of the average (over all 11 queries) query runtimes. As an overall net performance evaluation (based on overall average query runtime), QUETSAL-B4 is 15% better than FedX which in turn 10% better than ANAPSID. ANAPSID is 15% better than QUETSAL-B1 which is 11% better than SPLENDID+HiBISCUS.

An interesting observation is that QUETSAL-B4 ranked fourth in terms of the total query count for which one system perform better than others. However, it ranks first in terms of the average performance evaluation. The reason is that QUETSAL-B4 spend an extra 30ms (on average) for the source selection as compared to QUETSAL-B1 (ref. Table 17). The majority of the FedBench queries are simple in complexity, thus the query runtimes are very small (9/11 queries have runtime less than 750msec for QUETSAL). Consequently, 30msec makes a big difference in FedBench, showing the need of another federation benchmark which contains more complex queries.

The most interesting runtime results was observed for query LS6 of FedBench. QUETSAL-B4 only spends 643msec while QUETSAL-B1 spends 31sec for the same query (other systems also spend more than 5 seconds). A deeper look in to the query results shown that QUETSAL-B4 produces the maximum number of remote joins (i.e., 3) for this query. There was only a single join needs to be performed locally by the QUETSAL. This shows that one of the main optimization step in SPARQL endpoint federation is to generate maximum remote joins (loads transferring). The number of remote joins are directly related to the number of exclusive groups generated which in turn directly related to efficient triple pattern-wise source selection. Thus, QUETSAL was proposed with the main aim to produce maximum remote joins via efficient triple pattern-wise source selection. Finally, we looked in to the reason behind the three queries (LD6, LD6, LS5) results in runtime errors in QUETSAL. It was noted that QUETSAL SPARQL 1.1 query rewrite produces many single triple pattern SPARQL SERVICE clauses and their results are union together using SPARQL UNION clauses. Thus Sesame first retrieve all of the results from each of the relevant source of that triple pattern and then union them, resulting

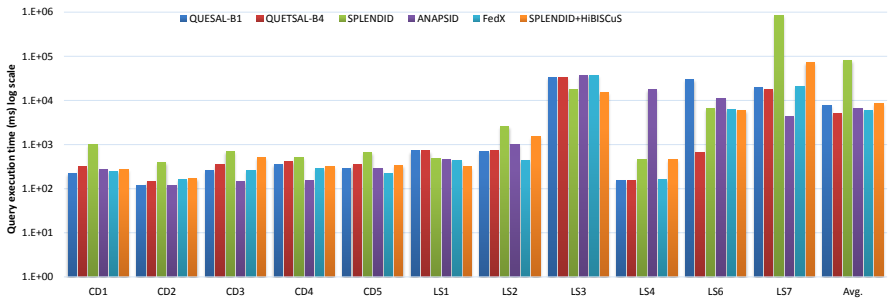


Figure 38: Query runtime evaluation.

in a to large number of intermediate results due to which a runtime error is thrown by the Sesame API. The runtime errors are provided at our project home. Finally, we believe that QUETSAL will perform more better in complex queries (containing many triple patterns) by producing many remote joins as well as on live SPARQL endpoints with Big Data where source overestimation is more expensive.

6

DUPLICATE-AWARE SOURCE SELECTION

The emergence of the Web of Data has resulted in a large compendium of interlinked datasets from multiple domains available on the Web. The central principles underlying the architecture of these datasets include the decentralized provision of data, the reuse of URIs and vocabularies, as well as the linking of knowledge bases [9]. Due to the independence of the data sources, certain pieces of information (i.e., RDF triples) can be found in multiple data sources. For example, all triples from the *DrugBank*[1] and *Neurocommons*[2] datasets can also be found in the *DERI health Care and Life Sciences Knowledge Base*[3]. We call triples that can be found in several knowledge bases *duplicates*.

While the importance of federated queries over the Web of Data has been stressed in previous work, the impact of duplicates has not yet received much attention. Recently, the work in [41] presented a benefit-based source selection strategy, where the benefit of a source is inversely proportional to the overlap between the source's data and the results already retrieved. The overlap is computed by comparing data summaries represented as Bloom filters [18]. The approach follows an "index-free" paradigm, and all the information about the sources is obtained at query time, for each triple pattern in the query.

In this chapter we present DAW, a duplicate-aware approach for federated query processing over the Web of Data. Similar to [41] our approach uses sketches to estimate the overlap among sources. However, we adopt an "index-assisted" approach, where compact summaries of the sources are pre-computed and stored. DAW uses a combination of min-wise independent permutations (MIPs) [20] and triple selectivity information to estimate the overlap between the results of different sources. This information is used to rank the data sources, based on how many new query results are expected to be found. Sources that fall below a predefined threshold are discarded and not queried.

We extend three well-known federated query engines – DARQ [71], SPLENDID [28], and FedX [95] – with DAW, and compare these extensions with the original frameworks. The comparison shows that DAW requires fewer sources for each of the query's triple pattern, therefore improving query execution times. The impact on the query recall due to the overlap estimation was minimal, and in most cases the recall was not affected. Moreover, DAW provides a source selection mech-

1 http://datahub.io/dataset/fu-berlin-drugbank
2 http://neurocommons.org/page/RDF_distribution
3 http://hcls.deri.org:8080/openrdf-sesame/repositories/hclskb

anism that maximises the query recall when the query processing is limited to a subset of the sources.

The rest of this chapter is organized as follows: Section 6.1 describes our novel duplicate-aware federated query processing approach. An evaluation of DAW against existing federated query approaches is given in Section 6.2.

6.1 DAW

In this section we present our DAW approach. DAW can be used in combination with existing federated query processing systems to enable a duplicate-aware query execution.

Given a SPARQL query q, the first step is to perform a *triple pattern-wise source selection*, i.e., to identify the set of data sources that contain relevant results for each of the triple patterns of the query. This is done by the underlying federated system. For a given triple pattern, the relevant sources are also called *capable* sources. The idea of DAW federated query processing is, for each triple pattern and its set of capable sources, to (i) *rank* the sources based on how much they can contribute with *new* query results, and (ii) *skip* sources which are ranked below a predefined threshold. We call these two steps *triple pattern-wise source ranking* and *triple-pattern wise source skipping*. After that, the query and the list of not skipped sources are forwarded to the underlying federated query engine. The engine generates the subqueries that are sent to the relevant SPARQL endpoints. The results of each subquery execution are then joined to generate the result set of q.

To better illustrate this, consider the example given in Figure 39, which shows a query with two triple patterns (t_1 and t_2), and the lists of capable sources for both patterns. For each source we show the total number of triples containing the same predicate of the triple pattern and the estimated number of new triples, i.e. triples that do not overlap with the previous sources in the list. The triple pattern-wise source ranking step orders the sources based on their contribution. As we see in the example, for the triple pattern t_1, source d_1 is ranked first, since it is estimated to produce 100 results. d_1 is followed by d_2, which can contribute with 40 new results, considering the overlap between the two sets. d_3 is ranked last, despite having more triples than d_2. This is because our duplicated-aware estimation could not find any triple in d_3 which is not in either d_1 or d_2. In the triple-pattern wise source skipping step, d_3 will be discarded, and t_1 will not be sent to d_3 during query execution. We can also set a threshold on the minimum number of results. For instance, by setting the threshold to 10 results, source d_4 will be skipped, since it can only contribute with 5 new results for t_2. By applying our duplicate-aware approach – which would select d_1 and d_2 both for *t1* and *t2*

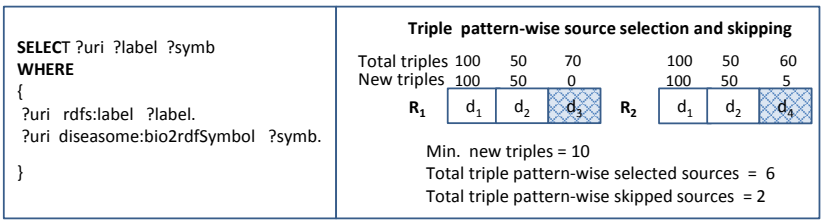


Figure 39: Triple pattern-wise source selection and skipping example

and would skip d_3 and d_4 – we would only send subqueries to two endpoints instead of four.

Both steps are performed prior to the query execution, by using only information contained in the DAW index. The main innovation behind DAW is to avoid querying sources which would lead to duplicated results. We achieve this by extending the idea of min-wise independent permutations (MIPs) [20], which are explained in the next section.

6.1.1 *Min-Wise Independent Permutations (MIPs)*

The main rationale behind MIPs is to enable the representation of large sets as vectors of smaller magnitude and to allow the estimation of a number of set operations, such as overlap and union, without having to compare the original sets directly. The basic assumption behind MIPs is that each element of an ordered set S has the same probability of becoming the minimum element under a random permutation. MIPs assumes an ordered set S as input and computes N random permutations of the elements. Each permutation uses a linear hash function of the form $h_i(x) := a_i * x + b_i \ mod \ U$ where U is a big prime number, x is a set element, and a_i, b_i are fixed random numbers. By ordering the set of resulting hash values, we obtain a random permutation of the elements of S. For each of the N permutations, the MIPs technique determines the minimum hash value and stores it in an N-dimensional vector, thus capturing the minimum set element under each of these random permutations. The technique is illustrated in Figure 40.

Let $V_A = [a_1, a_2, \ldots, a_N]$ and $V_B = [b_1, b_2, \ldots, b_N]$ be the two MIPs vectors representing two ordered ID's sets S_A, S_B, respectively. An unbiased estimate of the pair-wise resemblance between the two sets, i.e. the fraction of elements that both sets share with each other, is obtained by counting the number of positions in which the two MIPs vectors have the same number and dividing this by the number of permutations N as shown in Equation 2. It can be shown that the expected error in the estimation $O(1/\sqrt{N})$ [20]. Given the resemblance and the sizes of the two set, their overlap can be estimated as shown in Equation 3. A MIPs vector representing the union of the two sets, S_A and S_B, can be created directly from the individuals

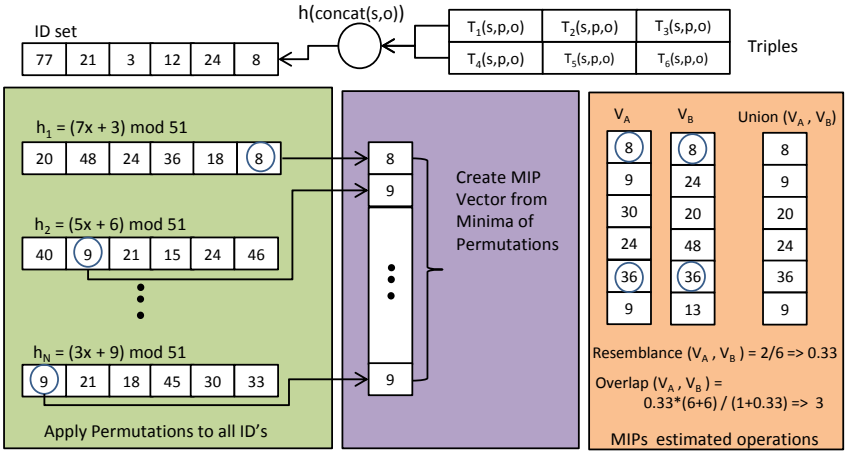


Figure 40: Min-Wise Independent Permutations

MIPs vectors, V_A and V_B, by comparing the pair-wise entries, and storing the minimum of the two values in the resulting union vector (see Figure 40). A nice property of MIPs is that unions can be computed even if the two MIPs vectors have different sizes, as long as they use the same sequence of hash functions for creating their permutations. In general, if two MIPs have different sizes, we can always use the smaller number of permutations as a common denominator. This incurs in a loss of accuracy in the result MIPs, but still yields to a more flexible setting, where the different collections do not have to agree on a predefined MIPs size [59].

$$\text{Resemblance}(S_A, S_B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|} \approx \frac{|V_A \cap V_B|}{N} \tag{2}$$

$$\text{Overlap}(S_A, S_B) \approx \frac{\text{Resemblance}(V_A, V_B) \times (|S_A| + |S_B|)}{(\text{Resemblance}(V_A, V_B) + 1)} \tag{3}$$

In the DAW index, MIPs are used as follow: For a distinct predicate p belonging to a data source d, we define $T(p, d)$ as the set of all triples in d with predicate p. A MIPs vector is then created for every $T(p, d)$. First an *ID set* is generated by mapping each triple in $T(p, d)$ to an integer value. A triple is given in the form of subject, predicate and object tuples, i.e. $< s, p, o >$. Since all triples in $T(p, d)$ share the same predicate by definition, the mapping is done by concatenating the subject (s) and object (o) of the triple, and applying a hash function to it (Figure 40). Then, the MIPs vector is created by computing the N random permutations of each element in the *ID set* and storing their minimum value. Finally, the MIPs vector is stored and mapped to each capability of the service description, as explained in the next section.

6.1.2 *DAW Index*

In order to detect duplicate-free subqueries, DAW relies on an index which contains the following information for every distinct predicate p in a source d:

1. The total number of triples $n_S(p)$ with the predicate p in d.
2. The MIPs vector $MIPs_S(p)$ for the predicate p in d, as described in the previous section.
3. The *average subject selectivity* of p in d, $avgSbjSel_S(p)$.
4. The *average object selectivity* of p in d, $avgObjSel_S(p)$.

The average subject and object selectivities are defined as the inverse of the number of distinct subjects and objects which appears with predicate p, respectively. For example, given the following set of triples:

$$S = \{< s_1, p, o_1 >, < s_1, p, o_2 >, < s_2, p, o_1 >, < s_3, p, o_2 >\} \tag{4}$$

the $avgSbjSel_S(p)$ is equal to $\frac{1}{3}$ and the $avgObjSel_S(p)$ is $\frac{1}{2}$. These two values are used in combination with the MIPs vector to address the expressivity of SPARQL queries as explained below.

Suppose that in a given triple pattern, neither the subject nor the predicate are bound. That means the pattern is of the form $<?s, p, ?o >$, where the question mark denotes a variable. In this case, the MIPs vectors in the DAW index can be used directly to estimate the overlap among the data sources that can provide results for the pattern. This is because the MIPs vectors are created by grouping triples according to their predicate. However, if any of the subject or object is bound (for example, $< s_1, p, ?o >$), the selectivity of the pattern becomes much higher and the MIPs vectors alone are unable to address this. As a result, overlap will be overestimated. To address this issue the modify Equation 3 to account for the subject and object selectivities as follows:

$$Overlap_t(S_A, S_B) \approx \frac{Resemblance(V_A, V_B) \times (|S'_A| + |S'_B|)}{(Resemblance(V_A, V_B) + 1)} \tag{5}$$

where the original size of a set S_i is replaced by a value $|S'_i|$ which is given by the following equation:

$$|S'_i| = \begin{cases} |S_i| & \text{if neither subject nor object are bound,} \\ |S_i| \times avgSbjSel_S(p) & \text{if subject is bound,} \\ |S_i| \times avgObjSel_S(p) & \text{if object is bound.} \end{cases}$$

```
[] a sd:Service ;
     sd:endpointUrl   <http://localhost:8890/sparql> ;
     sd:capability [
      sd:predicate  diseasome:name ;
      sd:totalTriples     147  ;
      sd:avgSbjSel  ''0.0068'' ;
      sd:avgObjSel  ''0.0069'' ;
      sd:MIPs  ''-6908232 -7090543 -6892373 -7064247 ...''; ] ;
     sd:capability [
      sd:predicate  diseasome:chromosomalLocation ;
      sd:totalTtriples     160  ;
      sd:avgSbjSel  ''0.0062'' ;
      sd:avgObjSel  ''0.0072'' ;
      sd:MIPs  ''-7056448 -7056410 -6845713 -6966021 ...''; ] ;
```

Listing 9: DAW index example

We call the set $C_S(p) = \{p, n_S(p), avgSbjsel_S(p), avgObjSel_S(p), MIPs_S(p)\}$ a *capability* of the data source. The total number of capabilities of a data source is equal to the number of distinct predicates in it.

It is crucial to keep the index size small to minimise the pre-processing time. On the other hand, this index must also contain sufficient information to enable an accurate source selection and duplicate-free subquery generation. Some federated query approaches such as DARQ and SPLENDID already provide the total number of triples, as well as the average selectivity values. Therefore, the storage overhead create by the DAW index depends mostly on the size of the MIPs vectors which can be adjusted to any length. In general, MIPs can provide a good estimation of the overlap between sets with a few integer in length. An example of a DAW index is given in Listing 9.

6.1.3 *DAW Federated Query Processing*

As explained earlier, given a SPARQL query, DAW performs the triple pattern-wise source ranking and skipping steps in order to rank the sources based on how much they can contribute with *new* query results, and skip sources which are below a given threshold. In this section we describe these two steps in detail.

Triple Pattern-wise Source Ranking: Given the heterogeneity and independence of data sources, it is expected that each source contributed differently in answering a given triple pattern, and the same result might be returned by multiple sources. Our goal is to provide a rank of the sources, according to the estimated number of new results it can contribute. By new results we mean with respect to the results already retrieved from sources ranked higher. The source ranking step works as follows: First, as no source has been ranked yet, the algorithm chooses the largest source, as it will likely to contribute with more results. To select the next source we use the DAW index to compute the estimated overlap between the already selected source and

every remaining source. The remaining source with the least amount of overlap is then chosen and ranked second. Before selecting the next source in the rank, we first need to estimate the union of the already selected sources. This is needed since we want to find out how much a source can contribute with results are not in the sources selected so far. The union can be easily estimated by applying a vector operator on the original MIPs, as explained in Section 6.1.1. The new union MIPs can be further combined with other MIPs to get the estimation of the union among several sets. The source ranking step continues until no more sources are left to be ranked.

Triple Pattern-wise Source Skipping: Given the rank of capable sources, the next step is to prune the rank, but skipping sources which cannot contribute with a minimum number of new results. This is done by setting a threshold, and pruning every source which falls below it. Since the total number of results depends on the triple pattern, the threshold is chosen in terms of the minimum percentage of new results a source can contribute. For instance, if the threshold is set to zero, DAW will aim at retrieving as much results as possible, while still skipping sources which cannot contribute with new results. Alternatively, the threshold can be set to higher values, in cases where the tradeoff between recall and number of sources queries is more important.

The pseudo code of the triple pattern-wise source ranking and skipping is given in Algorithm 6. It takes a triple pattern $t_i(s, p, o)$, its list of capable sources R_i, and the predefined threshold value as input and returned a ranked list of a subset of the capable source set Rw_i, $Rw_i \subseteq R_i$ as output. The ranked list and the MPIs with the union of the selected sources are initialised with the largest source. Lines 8-14 adjust the size of the dataset to reflect the subject or object selectivities, depending on the query. Lines 15-16 estimate the overlap and number of new triples. The source with the highest amount of new triples is then selected (Lines 17-19). The triple pattern-wise source skipping is done in Line 23 and sources ranked higher than the threshold are added to the final ranked list (Line 24). The union MIPs is then updated (Line 26) and the algorithm continues until no more sources are left.

Before we present our experimental analysis of DAW it is important to note the difference between the number of triple pattern-wise sources and the number of sources (e.g. SPARQL endpoints). The total number of triple pattern-wise selected sources for a query is calculate as follow: Let $NS_i \in \{1 \dots M\}$ be the number of sources capable of answering a triple pattern t_i where M is the number of available (physical) sources. Then, for a query q with n triple patterns, $\{t_1, t_2, \dots t_n\}$, the total number of triple pattern-wise sources is the sum of the sources for individual triple patterns, i.e. $\sum_{j=1}^{n} NS_j$. In the exam-

Algorithm 6 Triple pattern source-wise ranking and skipping

Require: $t_i(s,p,o) \in T$; R_i; thresholdVal //triple pattern t_i, capable data sources of t_i; Threshold Value

1: $rank_1Source$ = getMaxSizeSource(R_i, t_i) ; rnkNo = 1
2: unionMIPs = getMIPs($rank_1Source$, t_i) //get MIP vector for a tp of a source
3: Rw_i[rnkNo] = selectedSource
4: R_i = R_i - {selectedSource}
5: rnkNo = rnkNo+1
6: **while** $R_i \neq \emptyset$ **do**
7: selectedSource = null; maxNewTriples =0
8: **for** each $d_i \in R_i$ **do**
9: MIPs = getMIPs(d_i, t_i)
10: **if** *s* is bound in t_i **then**
11: MIPsSetSize =MIPsSetSize*getAvgSbjSel(d_i,t_i)
12: **else if** *o* is bound in t_i **then**
13: MIPsSetSize =MIPsSetSize*getAvgObjSel(d_i,t_i)
14: **end if**
15: overlapSize = *Overlap*(unionMIPs,MIPs)
16: newTriples = MIPsSetSize - overlapSize
17: **if** newTriples > maxNewTriples **then**
18: selectedSource = d_i
19: maxNewTriples = newTriples
20: **end if**
21: **end for**
22: curThresholdVal = maxNewTriples / unionMIPsSetSize
23: **if** curThresholdVal $\geqslant$ thresholdVal **then**
24: Rw_i[rnkNo] = selectedSource
25: selectedMIPs = getMIPs(selectedSource, t_i)
26: unionMIPs = *Union*(unionMIPs,selectedMIPs)
27: rnkNo = rnkNo+1
28: **end if**
29: R_i = R_i - {selectedSource}
30: **end while**
31: **return** Rw_i //ranked list of capable sources for t_i

ple from Figure 39, the number of sources is 4 (d_1, d_2, d_3, d_4) but the number of triple pattern-wise sources is equal to 6.

6.2 EXPERIMENTAL EVALUATION

In this section we present an experimental evaluation of the DAW approach. We first describe the experimental setup, followed by the evaluation results. All data used in this evaluation can be found at the project web page.[4]

6.2.1 *Experimental Setup*

Datasets: For our experiments, we used four different datasets. The Diseasome dataset contains diseases and disease genes linked by disease-gene associations. The Publication dataset is the Semantic Web Dog Food dataset and contains information on publications, venues and authors of publications. The Geo dataset resulted from retrieving the portion of triples from DBpedia that maps resources to their geo-coordinates. Finally, the Movie dataset is the RDF version of IMDB and contains amongst others a large number of actors, movies and directors. To simulate a federated scenario with fragmented datasets distributed across several sources, we partitioned each dataset in 10 slices and distributed the slices across 10 data sources (one slice per data source). Each data source is a Virtuoso-2012-08-02 SPARQL endpoint with the specifications given in Table 19.

To distribute the data across our 10 endpoints we defined a *discrepancy factor*, which controls the maximal size difference between the different slices.

$$\mathrm{discrepancy} = \max_{1 \leqslant i \leqslant M} |L_i| - \min_{1 \leqslant j \leqslant M} |L_j|, \tag{6}$$

where L_i stands for the i^{th} slice. The data is first partitioned randomly among the slices in a way that $\sum_i |L_i| = D$ and $\forall i \forall j\, i \neq j \rightarrow ||L_i| - |L_j|| \leqslant \mathrm{discrepancy}$. None of the existing benchmarks for federated query processing addresses the data duplication issue. Therefore, in order to add duplicates among slices, we randomly selected a number of slices and duplicated their contents across all remaining slices. For the DAW index, we use MIPs vectors of different sizes to better reflect the number of triples per predicate in each source. The sizes were chosen in a way that the overall index size is kept small. Table 18 presents an overview of the datasets, including the total number of triples and total size, the size of the DAW index, the index generation time, the discrepancy value among the 10 slices, the number of slices that were duplicated and their corresponding ID.

4 https://sites.google.com/site/DAWfederation/

Dataset	Number Triples	Dataset Size (MB)	Index Size (MB)	Index. Gen. Time (sec)	Discrepancy	No. Duplicated Slices	Duplicate Slice ID
Diseasome	91,122	18.6	0.17	4	1,500	1	10
Publication	234,405	39.0	0.24	6	2,500	1	10
Geo	1,900,006	274.1	1.63	133	50,000	2	5,8
Movie	3,579,616	448.9	1.66	201	100,000	1	2

Table 18: Overview of the datasets used in the experiments

EP	CPU(GHz)	RAM	Hard Disk
1	2.2, i3	4GB	300 GB
2	2.9, i7	16 GB	256 GB SSD
3	2.6, i5	4 GB	150 GB
4	2.53, i5	4 GB	300 GB
5	2.3, i5	4 GB	500 GB
6	2.53, i5	4 GB	300 GB
7	2.9, i7	8 GB	450 GB
8	2.6, i5	8 GB	400 GB
9	2.6, i5	8 GB	400 GB
10	2.9, i7	16 GB	500 GB

Table 19: SPARQL endpoints specification

Queries: We used three types of queries in our experiments: Single triple patterns queries (STP), star-shaped queries (S-1, S-2), and path-shaped queries (P-1, P-2, P-3). Single triple pattern (STP) queries consist of exactly one triple pattern in the query. Star-shaped and path-shaped queries are defined as in [35]. A S-k star-shaped query has one variable as subject and k joins, i.e., (k+1) triple patterns. An example of a S-1 star-shaped query is given in Figure 39. A P-k path-shaped query is generated by using the object of one triple pattern as subject in the next triple pattern, and it also contains (k+1) triple patterns. Previous work has shown that these query shapes are the most common shapes found in real-world RDF queries [64]. Our benchmark data consisted of 79 queries as shown in Table 20. Some query shapes could not be used on certain datasets due to the topology of the underlying ontology. For example, P-1 queries could not be sent to the Geo dataset since it only contained object properties. Each type a query was executed we used a random resource as subject or object, depending on the query type. The predicates of all queries are fixed.

Dataset	STP	S-1	S-2	P-1	P-2	P-3	Total
Diseasome	5	5	5	4	5	2	26
Geo	5	5	5	-	-	-	15
Movie	5	-	-	-	-	-	5
Publication	5	5	5	7	7	4	33
Total	20	15	15	11	12	6	79

Table 20: Distribution of query types across datasets

Federated Query Engines: We implemented our DAW approach on top of three different federated query engines: DARQ [71], SPLENDID [28], and FedX [95]. Both DARQ and SPLENDID already provide an index with some of the statistics needed in DAW. Therefore, we only needed to extend this index. For FedX, which is index-free, we added an index similar to the one in DARQ with our DAW extension. The underlying query execution mechanism remained the same.

Metrics: We compared the three federated approaches against their DAW extensions. For each query type we measured (i) the average number of triple pattern-wise sources that were skipped, (ii) the average recall, and (iii) the average query execution time. We did not consider the number of endpoints requests, as it depends on a number of factors, such as join type, block and buffer size, that vary across the different federated query processors. The threshold was initially set to zero, in order to maximise recall while querying fewer sources. All experiments were carried out in a machine with a 2.53GHz i5 processor, 4 GB RAM, and 500 GB hard disk. Experiments were carried out in a local network, so the network costs were negligible. After the first warm up run, each query type was executed 10 times and results were averaged.

6.2.2 *Experimental Results*

Triple Pattern-Wise Source Skipping: Table 22 shows the number of capable triple pattern-wise sources that were skipped by our approach, for each query type, as well as the recall. The total number of triple pattern-wise sources selected by the original systems is shown in brackets. The threshold was set to zero, which means that only sources that were estimated to returned no new results were pruned. We can see that DAW can effectively reduce the total triple pattern-wise selected sources, thus enable fewer subqueries federation. The highest gain was in the Diseasome dataset, where 214 sources were skipped in the DARQ approach, without affecting the recall. This corresponds to a decrease on the number of queried sources from 459 to 245. In other words, a full recall was achieved by querying only 53% of the available triple pattern-wise sources. In all cases except in the Geo dataset, the recall was not affected and all relevant results were retrieved. In the Geo dataset, the DAW index incorrectly pruned a small number of relevant sources, but the recall was still 99.99%. That means that DAW can deliver the same query results while querying much fewer sources. The source selection methods from FedX and SPLENDID return the same set of sources, therefore the number of skipped sources was the same for both. Moreover, they both use SPARQL ASK queries in the selection mechanisms, which leads to a better performance for STP queries. For example, consider the STP

```
SELECT ?title WHERE
{ www2008-chapter:103 pub:title ?title. }
```

Listing 10: A Single Triple Pattern (STP) query example

Dataset	STP	S-1	S-2	P-1	P-2	P-3	Total	Recall
Diseasome	14(35)	30(77)	40(107)	35(65)	65(125)	30(50)	214(459)	100%
Geo	22(40)	23(55)	37(101)	-	-	-	82(196)	99.99%
Movie	22(38)	-	-	-	-	-	22(38)	100%
Publication	9(30)	10(37)	15(86)	14(60)	21(120)	32(102)	101(435)	100%
Total	67(143)	63(169)	92(294)	49(125)	86(245)	62(152)	419(1128)	-

Table 21: Distribution of the triple pattern-wise source skipped by DAW extensions for threshold value 0

query given in Listing 10 where both the subject and predicate are bound. It is likely that a WWW2008 chapter with id 103 is found in only one data source but the property `pub:title` may be found in every source. As a result, FedX and SPLENDID will only select a single capable source while DARQ will select all sources containing that predicate.

Query Execution Time: For each dataset and query type, we measured the average query execution time in each of the federated query approaches and also in their DAW extension. Again, the threshold was set to zero and the average was over 10 queries. Figures 41, 42, and 43 show the results. We can see that DAW improves the query performance for most of the cases. For three of the datasets, Diseasome, Geo and Movie, DAW improved the query execution times of all federated systems tested, for all query types. The query performance in the Diseasome dataset showed the highest improvements. This is due to the large number of triple pattern-wise sources that were pruned. We can also see that if the number of skipped sources is low – as for the Publication dataset – the overhead in computing the sources overlap can be higher than the execution time saved by querying fewer sources, so the overall query execution time is worse. The overall performance is summarised in Table 23. We were able to improve the query execution time in DARQ by 16.46%, the SPLENDID by 11.11%, and FedX by 9.76%. For the Diseasome dataset, the improvement for the DARQ approach was 23.34%. These are aver-

Dataset	STP	S-1	S-2	P-1	P-2	P-3	Total	Recall
Diseasome	7(28)	30(77)	40(107)	35(65)	65(125)	30(50)	207(452)	100%
Geo	19(37)	23(55)	37(101)	-	-	-	79(193)	99.99%
Movie	15(31)	-	-	-	-	-	15(31)	100%
Publication	3(24)	10(37)	15(86)	14(60)	21(120)	32(102)	95(429)	100%
Total	44(120)	63(169)	92(294)	49(125)	86(245)	62(152)	396(1105)	-

Table 22: Distribution of the triple pattern-wise source skipped by DAW extensions for threshold value 0

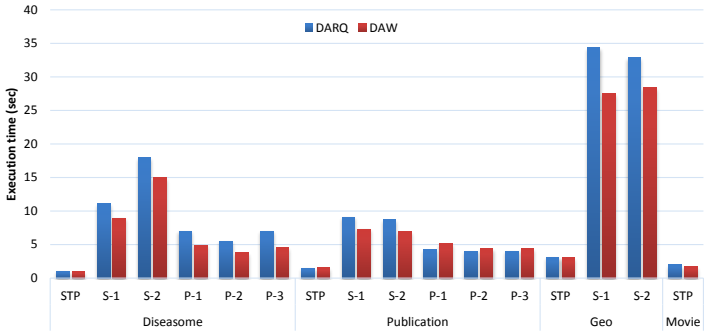


Figure 41: Query execution time of DARQ and its DAW extension

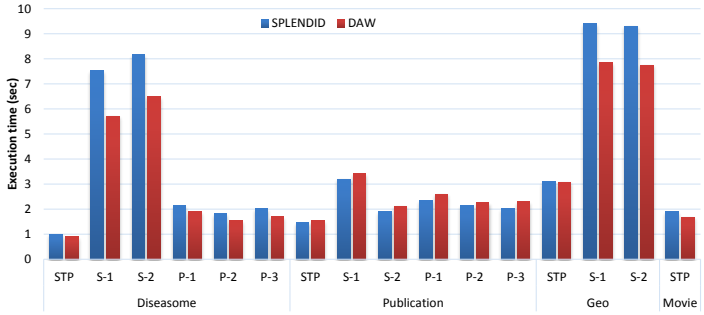


Figure 42: Query execution time of SPLENDID and its DAW extension

aged values across all datasets and query types. DAW led to a performance gain for most of the settings. We expect that in a setup with larger datasets and higher overlap, DAW can lead to even better improvements.

Number of Queried Sources vs. Query Recall:

The evaluation presented so far focused on achieving full recall, and only discarded sources that the DAW index estimated to contribute with no new results. We have shown that the estimation given by our algorithm is quite accurate, as only 0.01% of the results in one dataset were missing. There might be cases, however, where full recall is not crucial and the query processing budget is limited. Here, the goal is to retrieve as many results as possible by querying only

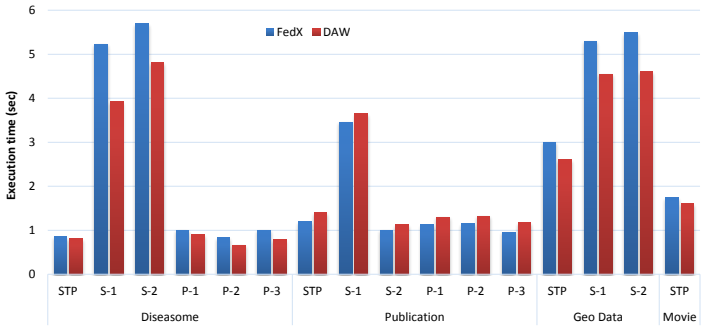


Figure 43: Query execution time of FedX and its DAW extension

	Diseasome		Publication		Geo Data		Movie		Overall	
	Exe.time	Gain	Exe.time	Gain	Exe.time	Gain	Exe.time	Gain	Exe.time	Gain
DARQ	8.27		5.26		23.44		1.96		9.59	
DAW	6.34	23.34	4.94	6.14	19.62	16.31	1.68	13.88	8.01	16.46
SPLENDID	3.78		2.18		7.27		1.90		3.71	
DAW	3.04	19.48	2.38	-8.94	6.22	14.40	1.68	11.16	3.30	11.11
FedX	2.44		1.48		4.60		1.74		2.44	
DAW	1.98	18.79	1.67	-12.38	3.92	14.71	1.61	7.59	2.20	9.76

Table 23: Overall performance evaluation. *Exe.time* is the average execution time in seconds. *Gain* is the percentage in the performance improvement

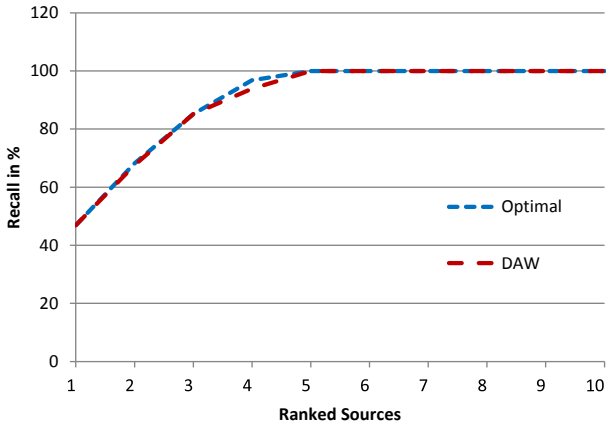


Figure 44: Diseasome: Recall for varied number of endpoints queried

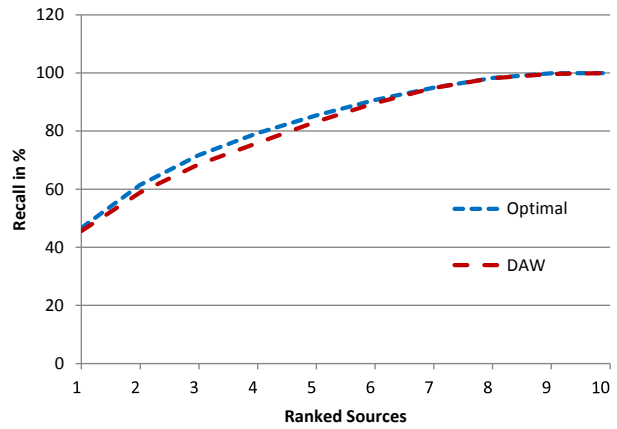


Figure 45: Publication: Recall for varied number of endpoints queried

a subset of capable sources. Standard federated query processing approaches are only able to identify the set of capable sources. They are not able to compare the contribution of the sources in order to identify which subset yields to a better recall. With DAW, an approximation of this contribution is provided by the ranking step. For any given threshold, DAW is able to provide the subset of capable sources that will deliver the best recall for that number of sources. To demonstrate this, we computed the query recall for different threshold values for the DAW DARQ extension. We ran each of the STP queries 10 times on the Diseasome and Publication datasets and averaged the results. We varied the threshold value in order to limit the query to a fixed number of endpoints and we computed the query recall based on the DAW source selection. We compared it with the optimal duplicate-aware approach, where sources were manually selected to maximise the recall. The results are show in Figure 44 and Figure 45. We can see that, in both cases, the source selection given by DAW is very close to the optimal case. Moreover, our experiment demonstrates the great potential in using source ranking for federated query processing. For the Diseasome dataset, by querying only 3 out of the 10 endpoints, DAW is able to retrieve 80% of the query results. A full recall is achieved with only 6 endpoints. This naturally depends on the degree of overlap, but nevertheless it shows promising results that should be further explored.

7

POLICY-AWARE SOURCE SELECTION

Inspired by the publication of hundreds of Linked Datasets on the Web, researchers have been investigating federated querying techniques to enable access to this decentralised content. Query federation aims to offer clients a single-point-of-access through which multiple distributed data sources can be queried in unison. In the context of Linked Data, various optimised query engines have been proposed that can federate multiple SPARQL interfaces [94; 27; 5; 70; 100].

However, in the Healthcare and Life Sciences (HCLS) domain – where data-integration is often vital – real-world data are often sensitive: strict ownership is granted to individuals working in hospitals, research labs, clinical trial organisers, *etc.* Therefore, the legal and ethical concerns on (i) preserving the anonymity of patients (or clinical subjects); and (ii) respecting data ownership through policy-based access control; are key challenges faced by the data analytics community working within the HCLS domain. In this chapter, we focus on point (ii): we investigate policy-based access-control over user-restricted resources residing at different clinical locations. The key challenges for federated querying are efficient source selection (i. e.determining which sources are (ir)relevant) and query planning (i. e.determining an efficient query execution strategy). However, federated engines often apply source selection at the level of individual endpoints, whereas in a controlled environment, a user may only have access to certain graphs within an endpoint. Considering a data-access layer on top thus adds unique challenges: source-selection should be more granular to enable effective data-access protocols, and should be policy-aware to avoid wasteful requests to unauthorised resources.

In this chapter, we present SAFE: a query federation engine that supports policy-based access to sensitive statistical data. SAFE is motivated by the needs of three clinical organisations who wish to enable *controlled* federation over statistical data owned and hosted by multiple clinical sites. To enable interoperability, these data are modelled as RDF Data Cubes with use of SDMX dimensions [21] and agreed-upon domain-specific vocabulary. SAFE extends upon the FedX engine [94] with two novel contributions: (1) GRAPH-LEVEL SOURCE SELECTION so as to enable graph-based access-control and (2) OPTIMISATIONS FOR FEDERATING STATISTICAL DATA given as RDF Data Cubes. With these modifications, we show that when compared with FedX, SAFE can (i) support more granular graph-level access control on top, and can (ii) efficiently reduce the query execution time when federating RDF

Data Cubes. It is important to note that no existing SPARQL query federation engine supports policy-aware access control over statistical clinical data sets. We argue that a specialised extension is required in general purpose query federation engines – like FedX – to address the specific challenges in combining statistical and distributed datasets with access restrictions.

The rest of the chapter is structured as follows: Section 7.1 discusses our motivational scenario where data from different clinical locations are required to be queried and aggregated. Section 7.2 presents the three stages of SAFE query processing. Section 7.3 presents evaluation of SAFE against internal and external data sets.

7.1 MOTIVATING SCENARIO

Our work is informed by the needs of three clinical organisations: University Hospital Lausanne (CHUV)[1], Cyprus Institute of Neurology and Genetics (CING)[2], and ZEINCRO[3]. These organisations wish to develop a platform for analysing clinical data across multiple clinical sites, which would allow for increasing the total number of patients that are included in each analysis, thus increasing the statistical power of conclusions related to biomarkers, effectiveness and/or side-effects of drugs or combinations of drugs, correlations between patient groups, *etc.* The ultimate goal is to enable the collaborative identification of new drugs and treatments while reducing the high costs associated with clinical trials.

use of linked data: With these goals in mind, the three clinical organisations mentioned are partners in the Linked2Safety EU project[4]. The goal of the project is to leverage Linked Data technologies to enable collaborative sharing of patient data between clinical organisations: to provide uniform access methods and controlled vocabularies that enable automated interoperability. The two main goals of the Linked2Safety project are (i) discovery of eligible patient data—also known as subject selection criteria—that can be recruited for a clinical trial from multiple clinical sites; and (ii) enabling multi-centre epidemiological studies enabling better understanding of relationships between pathological processes, risk factors, adverse events, and between genotype and phenotype [16]. However, although Linked Data technologies can help enable multi-site interoperability, the community largely focuses on datasets that can be made open to the public. In contrast, clinical data is often of an extremely

1 http://www.chuv.ch/
2 http://www.cing.ac.cy/
3 http:/www.zeincro.com/
4 http://www.linked2safety-project.eu/

sensitive nature and there is often strict legislation in place protecting the privacy of patients.

legal and ethical implications of patient privacy: According to EU Data Protection Directive 95/46/EC[5], clinical studies that involve patient-specific information must adhere to data-access restrictions that preserve patient anonymity; more specifically, a data access mechanism must ensure that patient identity cannot be discovered by any direct *or indirect* means using the dataset. Similar legislation exists in other jurisdictions. To avoid sharing of individual patient records, the Linked2Safety consortium has developed a data mining approach for transforming original clinical data into statistical summaries that may aggregate (or indeed redact) multiple dimensions of raw data.

The result is a set of anonymised data cubes whose dimensions correspond to insensitive clinical parameters without personal information [6]. The resulting multidimensional output contains sufficient granularity to quickly decide if the dataset is relevant for a given analysis – e. g.to understand the scale and dimensions of the data – and to perform high-level meta-analyses of aggregate data. Said data cubes are represented in a standard format – namely RDF Data Cubes per the recent W3C standard [74] – to enable interoperability (e. g.use of controlled vocabularies for dimensions) and to allow later use of Linked Data publishing/access methods.

Diabetes	BMI_Abnormal	Hypertension	Cases
0	0	0	11
1	0	1	26

CHUV – S1

Diabetes	BMI_Abnormal	Hypertension	Cases
0	0	0	40
1	0	1	50

CING – S2

Diabetes	BMI_Abnormal	Hypertension	HIV	Cases
0	0	0	0	30
1	0	1	0	60

ZEINCRO – S3

Diabetes	Smoking	Gender	Cases
0	0	0 (F)	90
1	0	1 (M)	120

CHUV – S4

Figure 46: Example data cubes published by `CHUV`, `CING` and `ZEINCRO`

However, although the data considered are aggregated and do not contain personal information from patients – thus preventing direct deanonymisation – indirect methods of deanonymisation are impossible to prevent [26]. Thus it is impossible to open a dataset and *fully* guarantee that it will not (indirectly) compromise patient anonymity. Likewise, if a (bio)medical dataset necessarily involves genetic data, there exist identifying markers by which patients can be directly deanonymised; thus genetic data can only be pseudoanonymised. Given such issues, in practice, sharing clinical datasets – even aggregate statistics – is often conducted under a strict legal framework between parties.

5 http://www.dataprotection.ie/docs/EU-Directive-95-46-EC/89.htm

In order to employ stricter data access restrictions on the anonymised multi-dimensional RDF data cubes, we then require a query federation approach that enforces and optimises for restricted user access over the statistical information contained in these data cubes. To illustrate and motivate, we now discuss a detailed example.

Figure 46 shows four sample datasets published by three different clinical sites. Each observation represents the total number of patients exhibiting a particular adverse event. For example, the `CHUV-S1` observations describe the total number of patients (in the **Cases** column) that exhibit a particular combination of three adverse events: **Diabetes**, (Abnormal) **BMI** and/or **Hypertension**. The value 0 or 1 indicates if the condition is present. For example, the second row in `CHUV-S1` represents the observations that there were 26 cases involving both **Diabetes** and **Hypertension** but without problematic **BMI**.

Once the data are published by clinical sites, they should be accessible to clinical researchers. Figure 47 shows a sample SPARQL query specifying subject-selection criteria, asking for the counts of cases that involve some combination of diabetes, abnormal BMI (Body Mass Index), and hypertension. An answer returned by the query, i. e.number of cases, will play a major role in deciding the resources (i. e.number of subjects, location, *etc.*) required for conducting a clinical trial. However, answering such a query requires integrating data cubes with three dimensions – **Diabetes**, **Hypertension**, **BMI** – and respective counts originating from multiple clinical sites.

Referring back to Figure 46, only three of the datasets (`CHUV-S1`, `CING-S2`, `ZEINCRO-S3`) contain all required dimensions. An answer returned by the query (Figure 47) should list counts (i. e.*cases*) from these 3 data cubes. However, assuming that the policy restrictions are applied to the user (say *James*), who wants to execute the query and has access to `CHUV-S1` and `CING-S2` data cubes only. Therefore, the query federation engine should retrieve results only from `CHUV-S1` and `CING-S2` and should not consider `ZEINCRO-S3` for querying.

```
PREFIX qb: <http://purl.org/linked-data/cube#>
PREFIX sehr: <http://hcls.deri.ie/l2s/sehr/1.0/>
SELECT ?diabetes ?bmi ?hypertension ?cases
WHERE { ?dataset a qb:DataSet.
        ?observation qb:dataSet  ?dataset;
        a qb:Observation;  sehr:Diabetes ?diabetes ;
        sehr:BMI_Abnormal ?bmi ;
        sehr:Hypertension ?hypertension ; sehr:Cases ?cases . }
```

Figure 47: Example subject selection criteria for clinical trials

Hence, one of the key requirements in the context of Linked2Safety project is to support federation of queries over clinical data distributed at multiple clinical sites by taking into account the data access policies (Figure 48 part c: shows a data access policy) assigned to the users (Figure 48 part a: shows a user profile for James) executing

those queries. Since data cubes are self-contained entities associated with additional provenance information (e. g.creator, location, *etc.*; see Figure 48 part b), they are modelled using named graphs [22] as supported in SPARQL.

In order to create clinical data cubes (Figure 46), formulate queries (Figure 47), describe user profiles, and their access rights (Figure 48) used within query federation process, the Linked2Safety consortium has developed two vocabularies: (i) *Semantic EHR Model (prefix "sehr")* describes the clinical terminologies used by the three clinical partners; and (ii) *Access Policy Model (prefix "lmds")* describes the user profiles (their activity, location, organisation, position and role) and their respective access rights (e. g.read, write). Considering the space limitation, further details of these two vocabularies are out of scope of this chapter where we refer the reader to Semantic EHR Model [75] and Access Policy Model [46] instead.

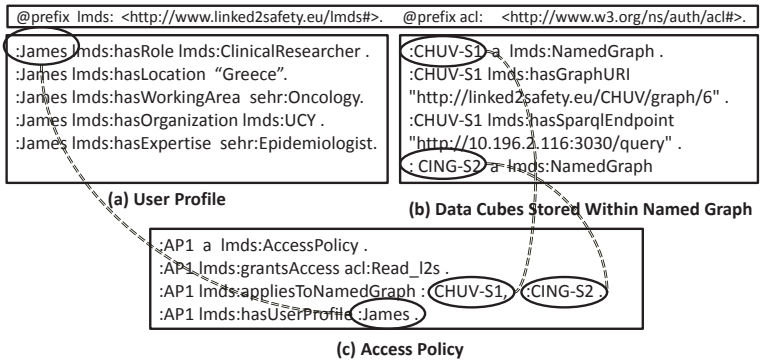


Figure 48: Snippets of user profile, access policy, conditions, and data cube storage

SAFE is uniquely designed to query statistical clinical data cubes (title of study, time period, the area, funding, etc.) and enforces restriction i. e.*denial* or *access* of these data cubes based on the description of user profiles (their activity, location, organisation, position and role), and their access rights.

7.2 METHODOLOGY AND ARCHITECTURE

SAFE's architecture is summarised in Figure 49 showing three main components: (i) Source Selection: performs multilevel source selection based on capabilities of data sources; (ii) Policy Aware Query Planning: filters the selected data sources based on a data-access right defined for each users; and (iii) Query Execution: performs the execution of subqueries against the selected sources, merging results returned. These components are described in detail in the following sub-sections.

source selection: SAFE performs a tree-based two level source selection as shown in the Figure 50. At Level 1, like other federa-

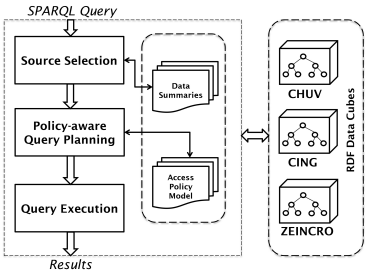


Figure 49: SAFE architecture

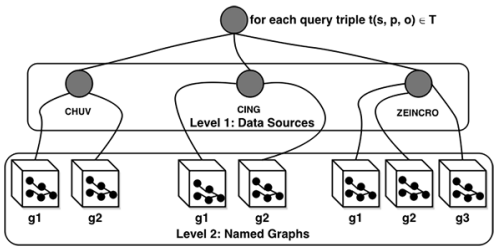


Figure 50: Tree-based two level source selection

tion engines [94; 27; 100; 1; 86], we do *triple-pattern-wise endpoint selection*, i. e.we identify the set of relevant endpoints that will return non-empty results for the individual triple pattern in a query. At Level 2 (unlike other federation engines), SAFE performs *triple-pattern-wise named graph selection*, i. e.we identify a set of relevant named graphs containing data cubes for all relevant endpoints already identified at the Level 1. SAFE relies on data summaries to identify relevant named graphs.

data summaries: We assume a set of datasets D where each dataset $d \in D$ is a SPARQL dataset: $d := \{(u_1, G_1), \ldots (u_n, G_n)\}$, where each (u_i, G_i) is a named graph with (unique) URI u_i and RDF graph G_i. In our use-case, named graphs refer to individual data cubes; we do not consider a default graph. We denote all graph names by $\text{names}(d)$ and a graph in the dataset by $d(u) := G$. We denote by $\text{preds}(G) := \{p \mid \exists s, o : (s, p, o) \in G\}$ the set of all distinct predicates in G and, $\text{preds}(d) := \bigcup_{(u,G)\in D} \text{preds}(G)$ the set of all distinct predicates in d. For each dataset $d \in D$, SAFE stores the following as a data summary: (i) the endpoint URL (`safe:endpointUrl`), where each endpoint indexes a dataset d; (ii) the set of all graph names in a dataset d: $\text{names}(d)$ where each graph contains a data cube (`safe:cube/safe:graph`); and (iii) for each graph $G \in d$, the set of all predicates in G: $\text{preds}(G)$ (`safe:cubeProperties`).

:source1	**lmds:endpointUrl**	<http://10.196.2.116:3030/query> .	CHUV
:source1	**lmds:cube**	:CHUV-s1 , :CHUV-s4 .	
:CHUV-s1	**lmds:graph**	<http:...CHUV-s1/graph/0007> .	
:CHUV-s1	**lmds:cubeProperties**	sehr:Diabetes, sehr:BMI_Abnormal, sehr:Hypertension, sehr:Cases .	
:CHUV-s4	**lmds:graph**	<http:...CHUV-s4/graph/0007> .	
:CHUV-s4	**lmds:cubeProperties**	sehr:Diabetes, sehr:Smoking, sehr:Gender, sehr:Cases .	
:source2	**lmds:endpointUrl**	<http://10.196.2.117:3030/query> .	CING
:source2	**lmds:cube**	:CING-s2 .	
:CING-s2	**lmds:graph**	<http:...CING-s2/graph/0007> .	
:CING-s2	**lmds:cubeProperties**	sehr:Diabetes, sehr:BMI_Abnormal, sehr:Hypertension, sehr:Cases .	
:source3	**lmds:endpointUrl**	<http://10.196.2.118:3030/query> .	ZEINCRO
:source3	**lmds:cube**	:ZEINCRO-s3 .	
:ZEINCRO-s3	**lmds:graph**	<http:...ZEINCRO-s3 /graph/0007> .	
:ZEINCRO-s3	**lmds:cubeProperties**	sehr:Diabetes, sehr:BMI_Abnormal, sehr:Hypertension, sehr:HIV, sehr:Cases .	

Figure 51: SAFE data summaries

We (informally) denote the set of all data summaries for d as $\mathcal{S}$ and the data summary for a particular source as $\mathcal{S}(d)$. An snippet of a data summary generated from the three data sources (`CHUV`, `CING`,

ZEINCRO) of Figure 46 is shown in Figure 51, where CHUV contains two data cubes (CHUV-S1, CHUV-S4), CING contains one data cube (CING-s2), and ZEINCRO also contain only one data cube (ZEINCRO-s3). Before explaining the source selection algorithm in the next section, we wish to make a formal description of the sets that are calculated on-the-fly by the source selection algorithm as part of the data summary:

1. The set of all predicates in a dataset d : $\text{preds}(d)$. This is the set-union of all $\text{preds}(G_i)$ for each $G_i \in d$. For example, $\text{preds}(\text{CHUV}) \coloneqq \{\text{Diabetes}, \text{BMI}, \text{Hypertension}, \text{Smoking}, \text{Gender}, \text{Cases}\}$ (ref. Figure 46, Figure 51).

2. The set of unique predicates in a data source d: $\text{upreds}(d) \coloneqq \{p \in \text{preds}(d) \mid \nexists d' \in D : d \neq d' \wedge p \in \text{preds}(d')\}$. For example, the unique predicates of CHUV: $\text{upreds}(\text{CHUV}) \coloneqq \{\text{Smoking}, \text{Gender}\}$ (ref. Figure 46, Figure 51).

3. The set of unique properties in a name graph with name u: $\text{upreds}(u, d) \coloneqq \{p \in \text{preds}(d(u)) \mid \nexists u' : u' \neq u \wedge p \in \text{preds}(d(u)')\}$ (overloading $\text{upreds}(.,.)$ for use with graphs also). For example, $\text{upreds}(\text{CHUV}{-}s1, \text{CHUV}) \coloneqq \{\text{BMI}, \text{Hypertension}\}$ and $\text{upreds}(\text{CHUV}{-}s4, \text{CHUV}) \coloneqq \{\text{Smoking}, \text{Gender}\}$ (ref. Figure 46, Figure 51).

4. The set of graphs in d with unique properties: $\text{unames}(d) \coloneqq \{u \in \text{names}(d) \mid \text{upreds}(u, d) \neq \emptyset\}$. For example, $\text{unames}(\text{CHUV}) \coloneqq \{\text{CHUV}{-}s1, \text{CHUV}{-}s4\}$.

SAFE's triple-pattern-wise source selection is shown in Algorithm 7. The algorithm is designed to exploit some specific properties of data cubes, particularly the locality of joins: assuming data cubes are not split over sources, certain types of joins can be answered locally. In particular, we define a subject–subject join (s–s join), where two triple patterns share (only) a subject variable, and a subject–object join (s–o join), where the join variable appears in the subject position of one triple pattern and the object of the other. For example, in Figure 47, triple patterns 1–2 form an s–o join and triple patterns 2–7 form an s–s join (in WHERE clause). As per the example, such joins would have to be answerable by one source/cube; thus (reasonably) assuming that data cubes are not split across sources, we can exploit this locality with a *join-aware strategy* that reduces sources considered relevant while ensuring complete results. For example, in Figure 47, though many sources will match the first triple pattern, they will not be considered relevant unless they are relevant for later triple patterns also.

algorithm: the source selection algorithm takes the set of all available sources D, their data summaries $\mathcal{S}$, the access policy P, the sender id $user$, and a SPARQL query containing a set of basic graph patterns[6] BGP as input (source selection only refers to BGPs, which may

6 http://www.w3.org/TR/sparql11-query/#BasicGraphPatterns

Algorithm 7 SAFE access policies-based triple pattern-wise source and named graph selection

Require: $D = \{d_1, \ldots, d_n\}$, $BGP = \{bgp_1, \ldots, bgp_m\}$, $\mathcal{S}$, P, user //sources, BGPs of a SPARQL query, SAFE summaries of sources, Access Policies, User

$\mathcal{R} \leftarrow \{\}$ //initialise relevance set
for each $bgp \in BGP$ **do**
  for each $t \in bgp$ **do**
    $R \leftarrow \{\}$ //initialise set of capable source graphs for triple pattern t
    $s \leftarrow subj(t)$, $p \leftarrow pred(t)$, $o \leftarrow obj(t)$
    //if predicate is bound
    if bound(p) **then**
      //for each data source in D
      for each $d \in D$ **do**
        $U \leftarrow upreds(d)$ //source unique properties read from $\mathcal{S}(d)$
        $A \leftarrow \bigcup_{d' \in D} upreds(d')$ //all unique properties read from $\mathcal{S}(d')$
        $E \leftarrow preds(D)$ //all properties of D
        if $p \in E \wedge (|bgp| = 1 \vee$ (StarJoin(t, bgp, U) $\vee$ PathJoin(t, bgp, U)) $\vee$ (!StarJoin(t, bgp, $A \setminus U$) $\wedge$!PathJoin(t,bgp,$A \setminus U$)) **then**
          triple pattern-wise named graph selection
          $UG \leftarrow \{\}$
          //for each unique cube
          for each $ug \in unames(d)$ **do**
            //unique p in bgp
            if $preds(bgp) \cap upreds(ug, d) \neq \emptyset$ **then**
              $UG \leftarrow UG \cup \{ug\}$ //ug might be relevant
            end if
          end for
          nothing unique so add all
          if $UG = \emptyset$ **then**
            $R \leftarrow R \cup (\{t\} \times names(d) \times \{d\})$
            //one unique cube
          else if $|UG| = 1$ **then**
            $R \leftarrow R \cup (\{t\} \times UG \times \{d\})$
          end if
          //if $|UG| > 1$, no source can match the join
        end if
      end for
    else if !bound(p) **then**
      //do triple-pattern-wise source selection using SPARQL ASK queries
      for each $d \in D$ **do**
        if ASK(d, t) = true **then**
          $R \leftarrow R \cup (\{t\} \times names(d) \times \{d\})$ //select all graphs
        end if
      end for
    end if
    //do policy-based graph filtering
    //for each capable graph
    for each $cg_i \in R$ **do**
      //Authorise user against graph of a source using access policies
      if ASK(d, cg_i, P, user) = false **then**
        RemoveGraph(cg_i, R) //remove unauthorised graph
      end if
    end for
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{R\}$
  end for
end for
return $\mathcal{R}$ //return relevant sources and graphs for triple patterns

be extracted from features such as `UNION` or `OPTIONAL`, etc.). The algorithm returns the set of relevant sources and corresponding named graphs for individual triple patterns as output. We handle the individual triple patterns of each BGP separately (*lines 1–2* of Algorithm 7). Given a triple pattern $t \in bgp \in BGP$ with bound predicate p, for each dataset $d \in D$, we collect the dataset-unique properties U, all unique properties A, and all dataset properties E from SAFE data summaries (*lines 3–10* of Algorithm 7). A dataset d is relevant for triple pattern t if its predicate is a set member of E and either t forms a star (s–s) or path join (p–o) with any other triple pattern (in the same query) having a predicate in U or t does not form both star and path join with set difference A/U (*line 11* of Algorithm 7). Once a relevant source is selected, the next step is to identify the set of relevant graphs within that relevant dataset (*lines 12–19* of Algorithm 7). If triple pattern t belongs to a unique graph ug in selected data source d then only uq is selected as relevant graph (*lines 13–15* of Algorithm 7). If there is no unique graph in d then all graphs in d are selected as relevant (*lines 16–19* of Algorithm 7). If a predicate is not bound in t, we fall back to a standard strategy and make use of SPARQL `ASK` queries for source selection, i. e.we send a SPARQL `ASK` request to each of the endpoints (*lines 20–23* of Algorithm 7). Once relevant graphs within relevant data sources are selected, the final step is to further prune the select capable graphs using policy-based filtering (*lines 24–26* of Algorithm 7). A capable graph cg is removed If query sender $user$ does not have grant access (according to policy policies P) for cg.

Per our running example, consider the triple pattern $tp :=$ `"?observation sehr:Diabetes ?diabetes"` of the query given in Figure 47. Since the predicate is bound in tp, the condition given at *line 6* of Algorithm 7 holds. For the `CHUV` dataset, U = `{Smoking, Gender}`, A = `{Smoking, Gender, HIV}`, and E = `{Diabetes, BMI, Hypertension, Smoking, Gender, Cases}` (*line 8–10*), and $A \setminus U$ = `{HIV}`. The predicate `Diabetes` $\in E$ and tp does not not form a star join (s–s) or path join (s–o) with $A \setminus U$ in the query. Therefore, the condition given at *line 11* of Algorithm 7 holds and the data source `CHUV` will be selected as relevant for TP. The next step is to select named graphs within the `CHUV` data source. For both of the named graphs (`CHUV-s1`, `CHUV-s4`) the condition given at *line 14* is true, therefore both named graph are selected as relevant. For both `CING` and `ZEINCRO` the condition given at *line 11* also hold; therefore they are also selected.

policy-aware query planning and query execution: After the identification of relevant sources for the SPARQL query, the next step is to further filter these sources by authenticating the user that is making the request. Policy-Aware Query Planning is the process of identifying *capable sources*: relevant sources that the user has ac-

```
PREFIX acl: <http://www.w3.org/ns/auth/acl#>
PREFIX lmds: <http://www.linked2safety.eu/lmds#>
ASK WHERE
?accessPolicy a lmds:AccessPolicy .
?accessPolicy lmds:appliesToNamedGraph ?namedGraph .
?namedGraph lmds:hasGraph lmds:CHUV_S1 .
?accessPolicy lmds:grantsAccess acl:Read_l2s .
?accessPolicy lmds:hasRequesterProle lmds:James  .
```

Figure 52: SPARQL query authenticating a user against a data cube/named graph

cess rights for. Access policies on each source are defined in the access policy model; for this, we consider graph-level control. The user authorisation is done by running SPARQL queries encoding information about the user and the relevant named graphs against the access-policy store. Considering the example discussed in Section 7.1, the SPARQL query generated for authenticating the user `James` for accessing the named graph `CHUV-S1` is shown in Figure 52. This query asks if there is any access policy that grants read access to the user `James` for the named graph `CHUV-S1`, returning `true` or `false`. As per the Figure 48 (part c), this example will return `true`.

Relevant named graphs that return `false` will be filtered. Endpoints with capable named graphs are then queried using standard federation techniques. For this, we use the FedX query engine [94], amending the query rewriter to append the capable graph information for each endpoint.

7.3 EVALUATION

This section presents evaluation comparing SAFE against FedX to validate the extensions we have proposed. The experimental setup (e. g.datasets, queries and metrics) for evaluation are as follows:

7.3.1 *Experimental Setup*

Datasets: We use two groups of datasets exploring two different use-cases.

The first group (INTERNAL DATASETS) are collected from the three clinical partners involved in our primary use-case as described in Section 7.1: `CHUV`, `CING` and `ZEINCRO`. These datasets contain aggregated clinical data represented as RDF Data Cubes and are privately owned/restricted.

The second group (EXTERNAL DATASETS) are collected from legacy Linked Data containing sociopolitical and economical statistics (in the form of RDF Data Cubes) from the `World Bank`, `IMF` (International Monitoring Fund), `Eurostat` and `Transparency International`. The `World Bank` data contains a comprehensive set of data about countries around the globe, such as observations on development indicators,

Table 24: Overview of Experimental Datasets

Dataset	Type	№ trip	№ obsv	№ sub	№ pred	№ obj	data	i.size	i.time
CHUV	INT	0.8 M	96 K	96 K	36	88	31 MB	-	-
CING	INT	0.1 M	17 K	17 K	21	51	5 MB	-	-
ZEINCRO	INT	0.4 M	49 K	49 K	24	59	15 MB	-	-
Total	INT	1.3 M	162 K	162 K	81	198	51 MB	8 KB	10 sec
World Bank	EXT	77 M	10 M	10 M	58	40 K	19 GB	-	-
IMF	EXT	18 M	1.8 M	1.8 M	30	3151	3.5 GB	-	-
Eurostat	EXT	0.3 M	38 K	44 K	31	5717	205 MB	-	-
Trans. Int.	EXT	43 K	3939	4286	64	5290	9.2 MB	-	-
Total	EXT	95 M	12 M	2 M	183	54 K	23 GB	12 KB	571 sec

financial statements, climate change, research projects, etc. The IMF data provides a range of time series data on lending, exchange rates and other economic and financial indicators. The Eurostat data provides statistical indicators that enable comparison between countries and regions across Europe. The Transparency International data includes a Corruption Perceptions Index (CPI), which ranks countries and territories based on how corrupt their public sector is perceived to be.

Table 24 gives an overview of the experimental datasets (**i.size** refers to index size and **i.time** to time taken for index generation). Each dataset was loaded into a different SPARQL endpoints (Jena Fuseki) on separate physical machines. **Queries:** A total of 12 queries were designed to evaluate and compare the federation performance of SAFE – for metrics such as source selection time, number of sources selected and total query execution – with those of FedX. These queries are of varying complexity and have varying type of characteristics. For space reasons, the full queries are available at http://linked2safety.hcls.deri.org:8080/SAFE-Demo/. In Table 25, we summarise the characteristics of these queries following similar dimensions to the SPARQL Berlin benchmark [17], showing their varying complexity. The number of sources counts those matched by at least one triple pattern.

Metrics: For each query type we measured (i) the number of sources selected; (ii) the average source selection time; (iii) the average query execution time; and (iv) the number of ASK requests issued to sources. The performance of SAFE and FedX was compared based on these metrics. All the experiments were carried out on a local network, so that network cost remains negligible. Machines used for experiments have a 2.60 GHz Core i5 processor, 8 GB of RAM and 500 GB hard disk running a 64-bit Windows 7 OS. The answers produced by FedX and SAFE were the same for all experiments.

Table 25: Summary of Query Characteristics

Characteristics/Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
# of Triple Patterns	9	7	9	16	7	8	11	10	7	7	3	7
# of Sources	3	4	4	3	4	3	3	4	3	3	3	3
# of Results	41	50	348	41	62	1983	5	10	1701	19656	570	41
Simple Filters												
Complex Filters							✓					
More than 9 pattens	✓		✓	✓	✓	✓	✓	✓				
Unbound predicates												
Negation							✓					
OPTIONAL operator												
LIMIT modifier		✓	✓				✓	✓				
ORDER BY modifier		✓	✓					✓				
DISTINCT modifier		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
REGEX operator					✓							
UNION operator	✓											
DESCRIBE operator												
CONSTRUCT operator												

7.3.2 *Experimental Results*

Triple pattern-wise sources selected: Table 26 shows the total number of triple pattern-wise (TP) sources selected by SAFE and FedX for all the queries. The last column shows the average number of TP sources selected by each approach. FedX performs optimal source selection at the triple-pattern-level using `ASK` queries for each triple pattern to find out precisely which sources can answer an individual triple pattern. By using join-aware source selection designed for RDF Data Cubes, SAFE manages to filter further potential sources that do not contribute to the end results, thus (as we will see) reducing response times.

Table 26: Sum of triple-pattern-wise sources selected for each query

System/Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Avg
SAFE	8	10	13	16	15	13	15	16	7	7	9	7	11
FedX	9	13	16	24	20	14	16	19	15	17	9	16	16

Number of SPARQL ASK requests: Table 27 shows the total number of SPARQL `ASK` requests used to perform source selection for each query. FedX is an index-free approach and performs runtime SPARQL `ASK` requests during source selection for each triple pattern in query. Conversely, SAFE uses data summaries for source selection, reverting to SPARQL `ASK` requests only when there is an unbound predicate in a triple pattern. None of our evaluation queries have an unbound predicate; hence there are no SPARQL `ASK` requests for SAFE. Though flexible in the generic case, index-free approaches can incur a large

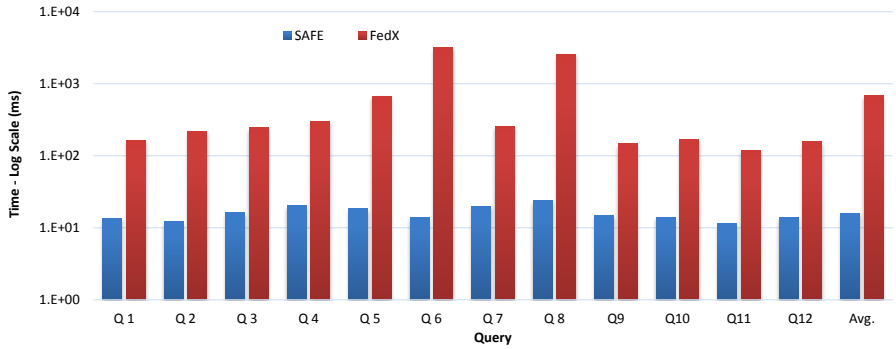


Figure 53: Comparison of source selection time

cost in terms of SPARQL `ASK` requests used for source selection, which can in turn increase overall query execution time.

Table 27: Number of SPARQL ASK requests used for source selection

System/Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Avg
SAFE	0	0	0	0	0	0	0	0	0	0	0	0	0
FedX	36	28	40	64	48	40	44	40	21	21	9	21	35

Source selection time: Figure 53 compares the source selection time of SAFE and FedX for all queries, where the y-axis is presented in log-scale. The rightmost pair of bars compares the average source selection time over all queries. As expected, the source selection time for SAFE is much less that of FedX: this is primarily attributable to SAFE's use of a domain-specific index for source-selection, which avoids incurring heavy traffic for `ASK` queries. The index can typically be pre-loaded into memory before query execution, which means that the source selection time for the presented use-case(s) will be minimal.

Query execution time: For each query, the average query execution time was calculated for both approaches by running each query ten times. Figure 54 compares the overall query execution time of SAFE and FedX for all queries. Again, the y-axis is logscale and the rightmost pair of bars compares the average query execution times. The results shows that SAFE has significantly outperformed FedX in all queries in the context of the presented use-cases. In fact, we see that FedX times-out in the case of three queries (in our experiments, we set queries to timeout after 25 minutes).

There are a number of factors that can influence the overall query execution time of a federation engine, such as join type, join order selection, block and buffer size, etc. However, given that SAFE is based on the FedX architecture, we can attribute the observed runtime improvements to three main factors: (i) source selection time is reduced (as we have seen in the previous sets of results); (ii) fewer sources are queried meaning less time spent waiting for responses; and (iii) triple patterns are more selective in SAFE, where, for ex-

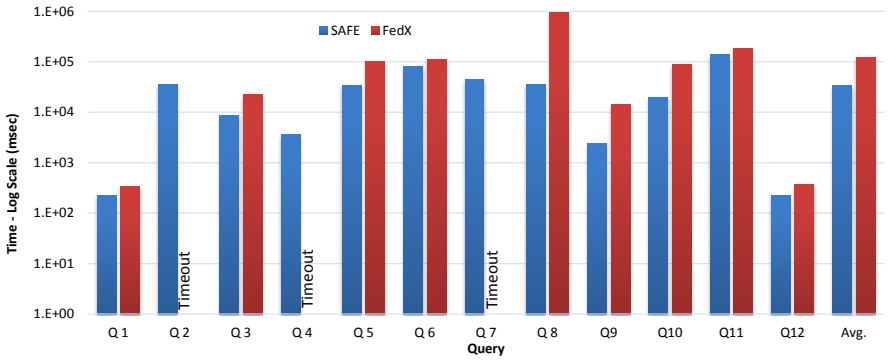


Figure 54: Comparison of query execution time

ample, our join-awareness makes it unlikely that all `rdf:type` triple patterns will need to be retrieved/queried for all sources but rather only from sources where such a triple pattern joins with a more selective one. Taken together, these three main observations explain the time saving observed for our presented use-cases.

8

DATA DISTRIBUTION-BASED SOURCE SELECTION

This chapter is based on [90] and explains TopFed, a personalized Cancer Genome Atlas[1] (TCGA) federation engine which keeps the data locality information in its index. The work is motivated by providing important biological queries, followed by the details of TCGA data conversion and distribution.

The TCGA is an effort led by the National Cancer Institute to characterize and sequence more than 30 cancer types from 9000 patients at the molecular level. The goal is to analyse DNA for every participant to discover abnormalities present in a tumour sample that are peculiar to the oncogenic process and whether it affect progression and regression of the tumours. Each cancer type published by TCGA has three levels. Level 1 is raw data, level 2 is normalized data, and level 3 is processed data. The analytics are performed on the level 3 data, which is also of our interest for the work presented in this chapter. TCGA is a valuable resource for hypothesis-driven translational research as all of its data results from direct experimental evidence. Analysis of such evidence within cancer research has led in recent years to clinically relevant findings in the genetic mark-ups of different cancer types and is at the forefront of a coordinated worldwide effort towards making more molecular results from cancer analyses publicly available [42].

Big data research initiatives such as the International Cancer Genomics Consortia[2], the 1000genomes[3] and the One Million Genomes project[4], the $10 Million Genome Prize[5], and the remarkable drop in the cost of genome sequencing[6] will soon mean that the current bioinformatics paradigm in which researchers download all the data, extract the interesting pieces and remove the rest, will no longer be feasible [51; 13]. The rapid development of advanced statistical methods for analysing cancer genomics [98; 10; 44] further emphasizes the need to enable smooth online data collection and aggregation. As pointed out in [23], "Large-scale genome characterization efforts involve the generation and interpretation of data at an unprecedented scale that has brought into sharp focus the need for improved information technology infrastructure and new computational tools to render the data suitable for meaningful analysis". A scalable and ro-

1 https://tcga-data.nci.nih.gov/tcga/
2 http://icgc.org/
3 http://www.1000genomes.org/
4 http://www.genomics.cn/en/navigation/show_navigation?nid=5658
5 http://in.reuters.com/article/2012/07/24/us-science-genome-prize-idINBRE86M02G20120724
6 http://www.genome.gov/sequencingcosts/

bust solution is therefore a critical requirement, whereby researchers can obtain a subset of big data they are interested in by executing a query using a particular service.

In addition to the large semi-structured experimental results available through TCGA and related projects, there is a significant number of unstructured and structured biomedical datasets available on the Web. Most of these datasets are critical towards annotating and integrating the experimental results. Remote query processing and virtual data integration, i.e., transparent on-the-fly-view creation for the end user, can provide a scalable solution to both challenges. Due to the majority of TCGA data being available in text files (in tabular format), it is difficult to query the contents of a particular file or to enable virtual data integration. In this chapter, we have addressed above problems by applying Semantic Web technologies and federated query processing. Semi-structured level 3 TCGA data were converted into Semantic Web standard format RDF such that it could be queried and publicly accessed via SPARQL endpoints. This choice of technology complies with the W3C recommendation of integrating distributed and heterogeneous data sources. There are currently a large number of applications supporting SPARQL and RDF, both academic and commercial, and both SwissProt[7] and EBI[8] have made their databases available as SPARQL endpoints.

In order to address the scalability issue while dealing with big data, we propose an efficient data distribution strategy and a TCGA tailored federated query engine (named TopFed) that leverages the data distribution along with the structure of triple pattern joins in a query for smart source selection. The logistics of the proposed solution will be assessed by comparison with a well established federation engine FedX [95].

8.1 MOTIVATION

Before TCGA, most cancer genomics studies have focused on only one type of data or one cancer histology. The Cancer Genome Atlas project changes that paradigm by making available to oncologists and biomedical scientists a comprehensive compilation of raw and processed data files on over 30 different cancer histologies and at several levels of "Genomics" (e.g. SNP, protein expression, exon expression, sequences, methylation, etc.). Since 2006, when the Cancer Genome Atlas first became available, multiple studies were devised to exploit its data. Nevertheless, a means to easily exploit this "cancer atlas" like one would exploit an atlas of planet Earth, does not yet exist. Part of the challenge is caused by a need to represent, or-

7 http://beta.sparql.uniprot.org/sparql
8 http://www.ebi.ac.uk/rdf/

ganize and structure the 28.3 TB of data[9] available to the public in a way that can be easily queried by computational/statistics tools. Further complicating this task has been the growth of TCGA data. Some institutions have access to the computational resources necessary to provide a TCGA-synchronized and query-able interface suitable to address the most complex questions such as comparing methylation across cancer histologies or correlating exon expression results with methylation patterns regardless of cancer histology. One institution providing a tool and query language to exploit this data is Memorial Sloan Kettering through its cBio portal[10]. However, the data must first be constrained to the type of cancer before it can be exploited from a biological/molecular stand point. A second challenge is caused by the applications of the data - not all data are useful for all cancer researchers. Some researchers focus on a particular type of data, or a particular cancer histology, and therefore have little or no interest in hosting the entire Cancer Genome Atlas in a structured, query-able form.

The aim behind the work presented in this chapter was to develop the computational concepts - and devise a prototype - that enable the exposure of TCGA as a distributed, semantically aware API (application programming interface). Although the data can be freely downloaded and analyzed by anyone with a sufficiently powerful computer, the computational tools available nowadays do not enable exploring this "atlas" without significant effort involved in selecting and downloading the data, mapping it to genomic coordinates and easily navigating to the sections of the genome that are relevant for understanding cancer. For example, zooming into genomic regions known as "Cancer Hotspots" or into the genomic coordinates where oncogenes and tumour suppressors are encoded, requires a combination of efforts including: 1) downloading the data; 2) parsing the text files for relevant results; and 3) mapping each file to the same set of genomic coordinates. On the other hand, fast, easy to use and integrated access to the big data such as TCGA requires: 1) Representing data in a format (e.g. RDF) amenable to integrated search; 2) logically connect all data; 3) distributing data across multiple locations (load balancing); and 4) supporting linking and federated querying (collecting data from more than one location using a single query) with external data sources.

TopFed is devised to address these requirements. Whereas requirements 1 and 2 are addressed using RDF and class level connectivity (see section TCGA Data Work flow), addressing requirements 3 and 4 relies on techniques that make the best use of the architecture of the Web to enable both redundancy when resources are down and sharing the load of hosting this data across multiple locations. As a proof

9 https://tcga-data.nci.nih.gov/datareports/statsDashboard.htm
10 http://www.cbioportal.org/

of concept, TopFed links different portions of the Cancer Genome Atlas across two institutions, one at Insight Centre for Data Analytics at NUIG in Ireland and other at the University of Alabama at Birmingham in United States. TopFed is devised as a federation query engine that enables selection of the appropriate endpoints necessary to address an incoming query as well as optimization of the services discovery based on metadata about each endpoint. TopFed accepts queries in SPARQL, the same universal, standardized query language as each of the endpoints connected to it, making its functionality straightforward. For example, if someone is looking to query only one cancer histology, they can direct their queries at the endpoint hosting that data. However, if someone wants to exploit and compare multiple cancer histologies, the query can be pointed at TopFed, which automates and optimizes the task of discovering endpoints that contain the data necessary to address the question. To illustrate a typical use case, we exemplify a genomic region query enabled by TopFed.

8.1.1 *Biological query example*

This example makes use of the KRAS gene, a gene that is commonly mutated and constitutively active in many cancer types, leading the cell to replicate DNA and make copies of itself at a very fast pace. Genes with this type of behaviour in the cell are commonly called oncogenes. When mutated, these genes become constitutive active, thus having the potential to cause normal cells to become cancerous. Imagine that for five different cancer histologies, we used TopFed to search for the methylation status of the KRAS gene (chr12:25386768-25403863), and created a box plot comparing the values, shown in Figure 55. The query (given in Listing 11) executed on each of the five SPARQL endpoints[11], resulting in five different samples.

This query returns the average methylation results for the KRAS gene of all patients in a particular cancer histology. The results show a clear distinction between solid tumours and hematopoetic cancers. This differential in the methylation values is not necessarily surprising results, given that blood cancers are known to be significantly different genetically from solid tumours. What is interesting and worth further exploring in these cases is the shape of the distribution: why Acute Myeloid Leukemia (AML) samples, a cancer of the myeloid of blood cells, appear to have high methylation, effectively creating a bi-modal distribution? Exploring the provenance of this data may provide a clue for that - one hypothesis is that these samples were incorrectly diagnosed as AML or it may be that these AML samples are indeed genetically different - and therefore should not be treated with the same therapies as the others. Since TopFed integrates both the

11 URLs of SPARQL endpoints hosting five cancer histologies that are shown in Figure 55 can be found at http://tcga.deri.ie/

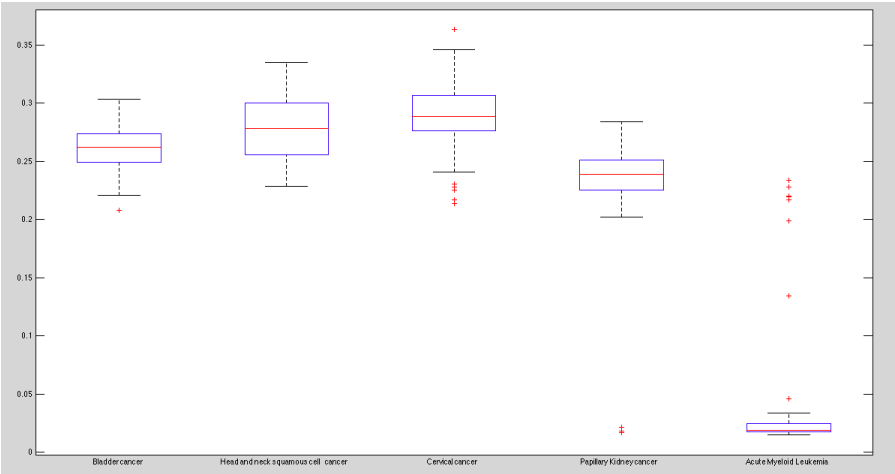


Figure 55: Biological query results. We used TopFed to search for the methylation status of the KRAS gene (chr12:25386768-25403863) across five cancer histologies (hosted by five SPARQL endpoints) and created a box plot comparing the methylation values. The corresponding SPARQL query to retrieve the required methylation values is given in Listing 11.

```
PREFIX tcga: <http://tcga.deri.ie/schema/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT DISTINCT ?patient avg(xsd:decimal(?methylationKRAS))
    as ?avgMethKRAS
WHERE
{
?patientURI tcga:bcr_patient_barcode ?patient .
?patientURI tcga:result ?recordNo .
?recordNo tcga:chromosome "12".
?recordNo tcga:position ?position.
?recordNo tcga:beta_value ?methylationKRAS.
FILTER (xsd:decimal(?position) >= 25386768 && xsd:decimal(?
    position) < 25403863 )
}
```

Listing 11: Query to retrieve average methylation values for the KRAS gene and for all patient of a particular cancer type

clinical and genomic parameters, exploring these different hypothesis is as easy as returning to the query and retrieving the potentially relevant clinical parameters that could explain the difference. Exploring the same gene (KRAS) in another type of data (e.g. exon expression) could also help explain why these samples are different. Since TopFed is "aware" of which SPARQL endpoints store each data property, it will appropriately select the correct source for the data, thereby adding the extra parameters to the query sufficient to generate sufficiently robust hypothesis.

Further exploring these examples is beyond the scope of this manuscript - however, we encourage our readers to experiment themselves with their own hypothesis or with a different set of genes/genomic locations by changing the values for tcga:chromosome and tcga:position. We include an example of a query that could be used to retrieve the clinical parameters for the outlier patients (and compare with non-outlier patients) in Listing 12.

The main contributions of this chapter are the following:

1. We have proposed a Linked Data version of TCGA that supports efficient data distribution and federated SPARQL queries to integrate data from multiple SPARQL endpoints efficiently by only sending remote queries.

2. We have published, to the best of our knowledge, the largest RDF dataset (20.4 billion triples) and linked it to various datasets in the LOD cloud to enable annotation and enhancement with public knowledge bases as well as virtual data integration.

3. We devised the basic architecture and logic rules governing TopFed, a smart federated query engine for virtual integration of the TCGA data from multiple SPARQL endpoints that comply with the TCGA organizational schema. Further, we provide easy to use utilities[12] in order to refine and transform TCGA raw text files into RDF.

4. We evaluate our approach against FedX using 10 different SPARQL queries and show that our source selection algorithm, on average, selects less than half sources compared to FedX (with 100% recall). Also, our average query processing time is one third in comparison to FedX.

The remaining part of this chapter is organized as follows: we present our methodology to refine, RDFize and link the TCGA data to LOD datasets in detail. Subsequently, we present a thorough evaluation of our approach against state of the art approaches. We finally conclude the chapter with a discussion of our findings and an overview of future work.

12 http://goo.gl/rtwm6q

```
PREFIX tcga: <http://tcga.deri.ie/schema/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?outliersPatient ?avgMethKRAS ?piordiagn ?vitalstat ?
    ageatdiag ?gender ?pretreatHistory ?ethnicity ?race
{
 {
SELECT DISTINCT ?outliersPatient (avg(xsd:decimal(?
    methylationKRAS)) as ?avgMethKRAS) ?piordiagn ?vitalstat ?
    ageatdiag ?gender ?pretreatHistory ?ethnicity ?race
 WHERE
 {
 ?patientURI <http://tcga.deri.ie/schema/bcr_patient_barcode>
     ?outliersPatient .
 ?patientURI <http://tcga.deri.ie/schema/result> ?recordNo .
 ?recordNo tcga:chromosome "12".
 ?recordNo tcga:position ?position.
 ?recordNo tcga:beta_value ?methylationKRAS.
 FILTER ( xsd:decimal(?position) >= 25386768 && xsd:decimal(?
     position) < 25403863 )
 SERVICE <http://vmlion14.deri.ie/node42/8081/sparql>
  {
 ?patientURI tcga:bcr_patient_barcode ?outliersPatient.
 ?patientURI tcga:prior_diagnosis ?piordiagn .
 ?patientURI tcga:vital_status ?vitalstat .
 ?patientURI tcga:age_at_initial_pathologic_diagnosis ?
     ageatdiag .
 ?patientURI tcga:gender ?gender .
 ?patientURI tcga:pretreatment_history ?pretreatHistory .
 ?patientURI tcga:ethnicity ?ethnicity .
 ?patientURI tcga:race ?race .
  }
 }
}
 FILTER (?avgMethKRAS > 0.05)
}
```

Listing 12: Query to retrieve average methylation values for the KRAS gene, along with clinical data, for all AML outlier patients. This query can be run at http://vmlion14.deri.ie/node45/8082/sparql

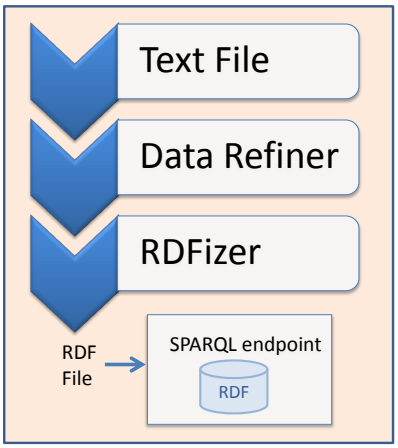


Figure 56: TCGA text to RDF conversion process. Given a text file, first it is refined by the Data Refiner. The refine file is then converted into RDF (N3 notation) by the RDFizer. Finally, the RDF file is uploaded into a SPARQL endpoint.

8.2 METHODS

8.2.1 *Transforming TCGA data to RDF*

The process of transforming TCGA data into RDF[13] is shown in Figure 56. Given a TCGA text file, the first processing step is carried out by the Data Refiner, which selects the specific fields[14] necessary for traditional molecular analysis algorithms. This step is necessary to minimize the size of the resulting RDF according to what we expect will be the most useful results. It is important to note that the above required fields for different types of results may not be directly accessible through raw text files. To this end, our Data Refiner makes use of the annotations files[15] for the required fields lookup. For example, methlylation annotation files are used to obtain chromosome and position values using Probe_Name lookup. Finally, the refined text file is sent to the RDFizer, which generates the resulting RDF dump in N3 format[16]. Our choice of N3 was due of its efficient space consumption. The generated RDF dumps[17] are then uploaded to various SPARQL endpoints according to the distribution rules shown in Figure 57.

An example of the above RDFication process is shown in Figure 58, where part of raw methylation result of patient TCGA-A2-AoCX is provided as input to the Data Refiner. The Data Refiner selects *chrome*, *position*, and *beta_value* out of the five available columns. The selected columns are commonly used for traditional molecular analysis algorithms targeting methylation data. It is important to note that Data

13 A step-by-step user manual is also available at: http://goo.gl/0oTAKV
14 https://code.google.com/p/topfed/wiki/SelectedFields
15 http://goo.gl/pb3o2G
16 http://www.w3.org/TeamSubmission/n3/
17 Available to download from: http://tcga.deri.ie/dumps/

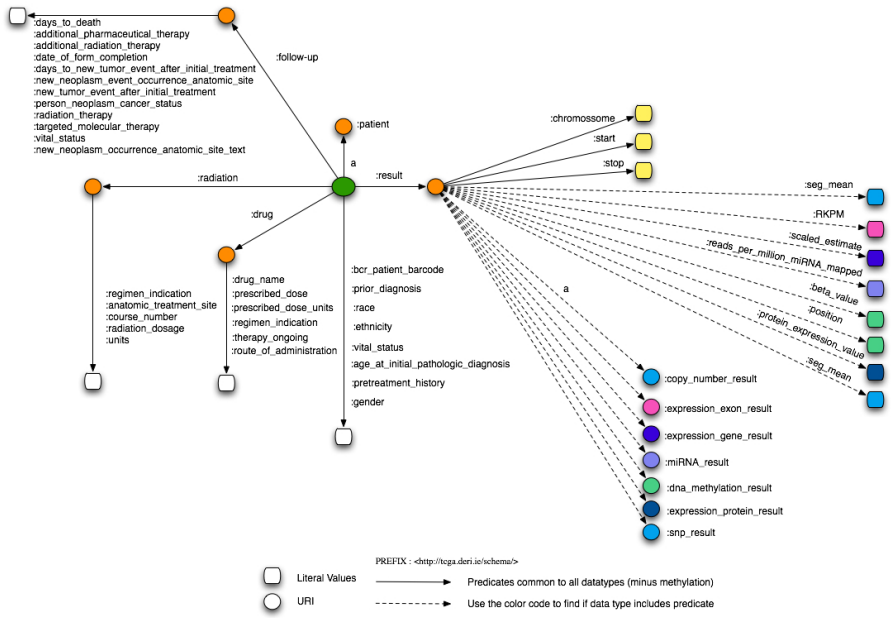


Figure 57: TCGA data distribution/load balancing and source selection. The proposed data distribution and source selection diagram for hosting the complete Linked TCGA data.

Refiner also skipped the yellow highlighted line because *beta_value* is not available for that specific methylation result. The refined text file is then passed to RDFizer that generates the RDF dump (N3 format). The values d1...d8 show DNA methylation results from 1 to 8. The use of this information is further explained in the Source Selection sub-section.

The accuracy of the text to RDF conversion is 100% (to the best of our understanding) since our Data Refiner selects a predefined set of fields for different types of results. Further, it skips specific field values (such as *NA, Null, Unknown, Not Reported etc.*) during RDFication process as shown in the above example. Currently, we have RDFized 27 cancer tumours and the statistics are shown in Table 28. We will RDFize new TCGA data once it is available through the TCGA data portal.

8.2.2 *Linking TCGA to the LOD cloud*

One of the design principles of Linked Data[18] is the provision of links to other data sources. Adding links from TCGA to other knowledge bases is particularly crucial to ensure that the information already contained in other data sources can be easily (1) merged with the new TCGA data as well as (2) queried in combination with the TCGA data by means of federated SPARQL queries[19]. Moreover, links can facili-

18 http://www.w3.org/DesignIssues/LinkedData.html

19 See http://www.w3.org/TR/sparql11-query/ for more information on federated queries based on SPARQL 1.1.

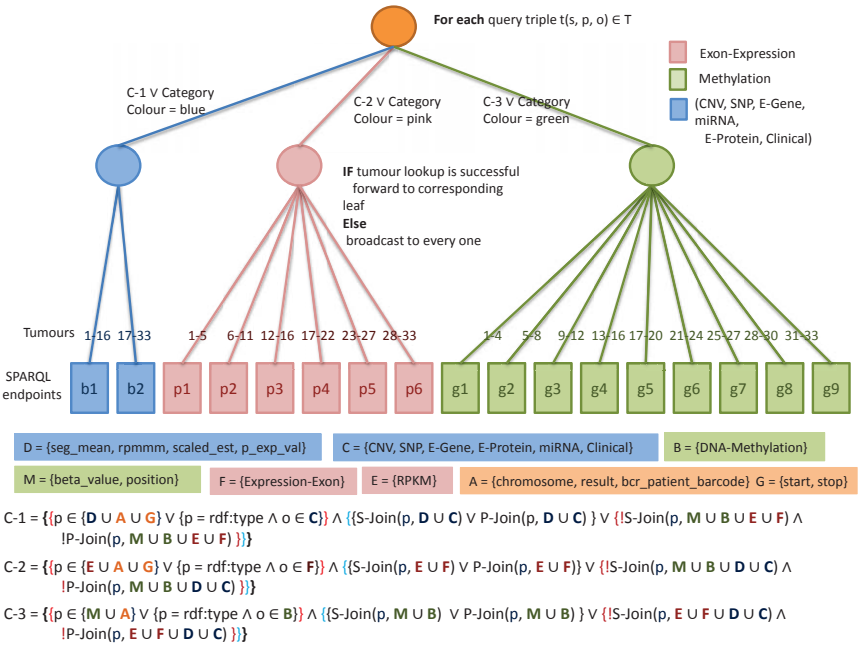


Figure 58: Text to RDF conversion process example. An example showing the refinement and RDFication of the TCGA file.

Table 28: **Statistics for 27 tumours sorted by number of triples**

Tumour Type	Raw(GB)	Refined(GB)	RDF(GB)	Triples(Million)
Lymphoid Neoplasm Diffuse Large B-cell Lymphoma (DLBC)	0.37	0.20	0.83	35
Cutaneous melanoma (UCS)	1.2	0.64	2.6	113
Glioblastoma multiforme (GBM)	2.3	0.77	2.8	132
Esophageal carcinoma (ESCA)	1.5	0.88	3.4	149
Adrenocortical carcinoma (ACC)	1.6	0.90	3.6	158
Pancreatic adenocarcinoma (PAAD)	2.6	1.1	4.5	200
Kidney Chromophobe (KICH)	3.7	1.4	5.3	242
Sarcoma (SARC)	3.8	1.5	5.9	267
Cervical (CESC)	8.75	2.44	8.86	400.19
Ovarian serous cystadenocarcinoma (OV)	8.2	2.4	8.7	410
Rectal adenocarcinoma (READ)	8.07	2.25	9.04	413.31
Papillary Kidney (KIRP)	10.40	2.90	10.4	469.65
Stomach adenocarcinoma (STAD)	5.5	2.9	12	529
Liver hepatocellular carcinoma (LIHC)	8.2	3.1	12	550
Bladder cancer (BLCA)	12.16	3.39	12.3	556.38
Acute Myeloid Leukemia (LAML)	14.85	4.14	15.1	684.05
Lower Grade Glioma (LGG)	17.08	4.76	17.1	778.82
Prostate adenocarcinoma (PRAD)	18.05	5.03	18.1	821.01
Lung squamous carcinoma (LUSC)	20.63	5.75	20.5	927.08
Cutaneous melanoma (SKCM)	23.22	6.47	23.2	1050.94
Uterine Corpus Endometrial Carcinoma (UCEC)	13	5.98	24.2	1070
Colon adenocarcinoma (COAD)	18	6.64	26	1175
Head and neck squamous cell(HNSC)	27.6	7.69	27.5	1245.37
Lung adenocarcinoma (LUAD)	23	9.1	36	1611
Kidney renal clear cell carcinoma (KIRC)	24	9.4	37	1658
Thyroid carcinoma (THCA)	26	10.1	40	1796
Breast invasive carcinoma (BRCA)	45	17	65	2959

A total of 20.4 Billion triples

Table 29: **Excerpt of the links for the lookup files of TCGA**

Source	Target	Class	# links	Runtime (ms)
DNA27	HGNC	Genes	23,181	154
DNA27	Homologene	Genes	27,654	193
DNA27	OMIM	Genes	15,171	158
DNA450	Homologene	Genes	489,643	5,710
DNA450	OMIM	Genes	212,284	429
DNA27	HGNC	Chromosomes	108,662	96
DNA27	OMIM	Chromosomes	16,039,535	8,055

The source column shows the name of the look-up file that was linked to the target dataset named in the second column. The class column shows the type of resources that were linked. The fourth column shows the number of links that were generated while the runtime column shows the time required by LIMES to carry out the linking process in ms.

tate other tasks such as cross-ontology question answering, data integration and data analytics. Yet, the sheer size of bio-medical knowledge base available on the LOD cloud and of the TCGA knowledge base itself makes it impossible to use manual linking to provide such cross knowledge-base links from TCGA to other data sources. We thus made use of the LIMES framework [65] for discovering links between TCGA and other knowledge bases. LIMES is a framework for link discovery that provides time-efficient implementations of several string and numeric similarity and distance measures. The framework provides both means to define link specifications explicitly and machine-learning algorithms for finding link specifications in an unsupervised and supervised fashion. Given that genes and chromosomes have dedicated IDs that are used across several biomedical knowledge bases, we used LIMES exactMatch measure for linking. We focused on linking patient data and lookup data with knowledge bases that describe genes and chromosomes. In particular, we linked TCGA to HGNC[20], OMIM[21] and Homologene[22]. Tables 29 and 30 provide an excerpt of the links generated for the TCGA dataset, while Listing 13 provides an excerpt of the specifications used for linking. The linking tasks were carried out on one kernel of a 2.3GHz i7 processor with 4GB RAM. Given that we used exact matches, we ensured that our link discovery achieves a precision of 100%. The recall of the linking process is tedious to assess as it would require assessing millions of links manually.

20 http://hgnc.bio2rdf.org/sparql
21 http://omim.bio2rdf.org/sparql
22 http://homologene.bio2rdf.org/sparql

```
<SOURCE>
 <ID>TCGA</ID>
 <ENDPOINT>dna_methylation450_Lookup.nt</ENDPOINT>
 <VAR>?x</VAR>
 <PAGESIZE>-1</PAGESIZE>
 <RESTRICTION>?x rdf:type tcga-schema:dna_methylation450_lookup</
     RESTRICTION>
 <PROPERTY>tcga-schema:Gene_Symbol AS lowercase</PROPERTY>
 <TYPE>N-TRIPLE</TYPE>
</SOURCE>
<TARGET>
 <ID>homologene</ID>
 <ENDPOINT>http://homologene.bio2rdf.org/sparql</ENDPOINT>
 <VAR>?y</VAR>
 <PAGESIZE>10000</PAGESIZE>
 <RESTRICTION>?y a homologene:HomoloGene_Group</RESTRICTION>
 <PROPERTY>homologene:has_gene_symbol AS lowercase</PROPERTY>
</TARGET>
<METRIC>exactmatch(x.tcga-schema:Gene_Symbol,
   y.homologene:has_gene_symbol)</METRIC>
<ACCEPTANCE>
 <THRESHOLD>1</THRESHOLD>
 <FILE>dna_450_homologene_accepted.nt</FILE>
 <RELATION>tcga-schema:Homologene</RELATION>
</ACCEPTANCE>
```

Listing 13: Excerpt of the LIMES link specification for linking TCGA and Homologene

Table 30: **Excerpt of the links for the methylation results of a single patient**

Source	Target	Class	# links	Runtime (ms)
Methylation	HGNC	Chromosomes	97,530	205
Methylation	OMIM	Chromosomes	14,407,269	6,095
Gene expression	HGNC	Chromosomes	86,052	80
Gene expression	OMIM	Chromosomes	12,535,829	4,679

The source column shows the name of the patient file that was linked to the target dataset named in the second column. The class column shows the type of resources that were linked. The fourth column shows the number of links that were generated while the runtime column shows the time required by LIMES to carry out the linking process in ms.

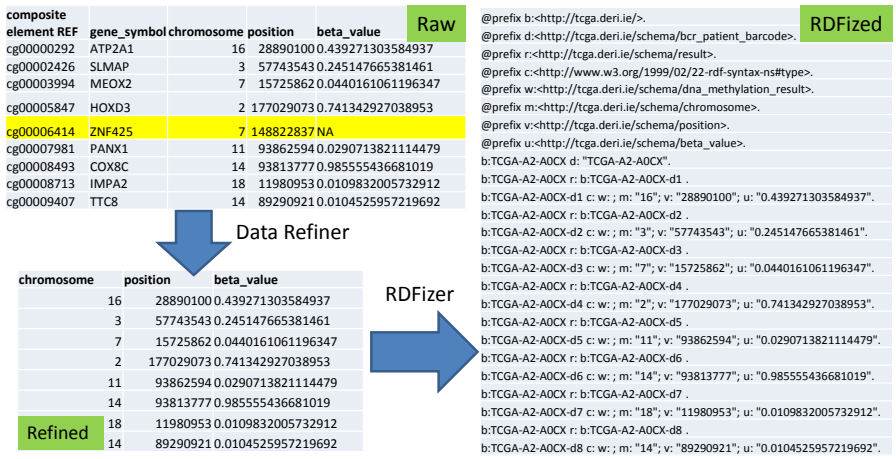


Figure 59: TCGA class diagram of RDFized results. Each level 3 data is further divided into three layers where: layer 1 contains patient data, layer 2 consists of clinical information and layer 3 contain results for different samples of a patient.

8.2.3 *TCGA data workflow and schema*

To devise a fast, big data driven query federation engine, we started by exploiting how the various files and types of data in TCGA are interconnected. To date, 23054 raw data files from 28 cancer tumours have been collected, summing up to a total of 28.3 TB of data[23]. For each level 3 data, we have identified three different types, i.e., we RDFized level 3 data for each cancer type and further define 3 data types for each of the level 3 tumours data of data. The resulting data are organized as a three layer architecture where layer 1 contains patient data, layer 2 consists of clinical information and layer 3 contain results for different samples of a patient. Each type of data is assigned to a different class in the RDFized version as depicted in Figure 59. For each patient, tumour and blood/normal tissue samples are collected and divided into different portions upon which different protocols such as DNA, RNA and so on, are applied to extract the analytes for the analysis of the sample. The extracted analytes are distributed across plates. All these plates containing patients tumour and normal samples are shipped to Genome Characterization Centres (GCCs) and Genome Sequencing Centres (GSCs) that produce different data type results which are shown in layer 3 (cf. Figure 59). The schema of the corresponding Linked TCGA is shown in Figure 60. We have included only important properties from clinical data (e.g., drug, follow-up, radiation etc.) as the complete list of properties is around 300. This diagram is useful to understand the connectivity between the Linked TCGA data and to formulate SPARQL queries.

23 https://tcga-data.nci.nih.gov/datareports/statsDashboard.htm

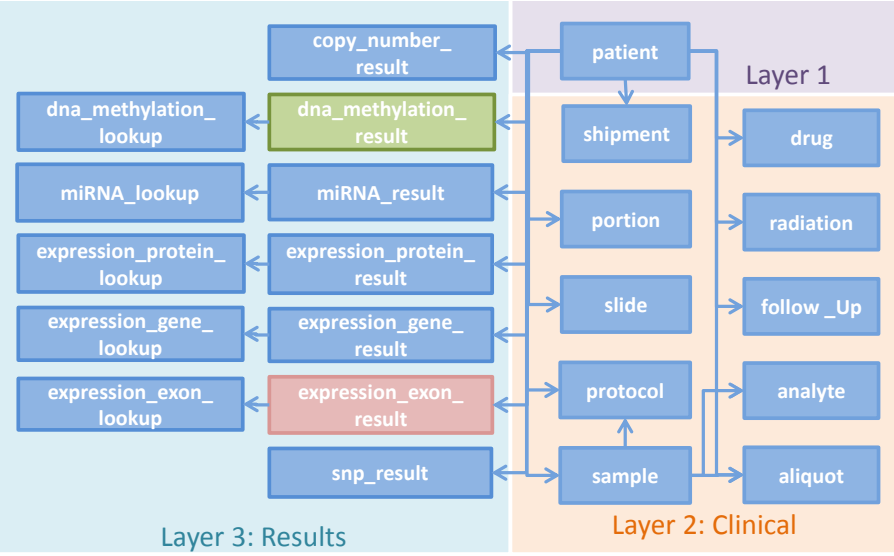


Figure 60: Linked TCGA schema diagram. The schema diagram of the Linked TCGA, useful for formulating SPARQL queries.

8.2.4 *Data distribution and load balancing*

A key property of the federation method described here is the efficient distribution of the data among SPARQL endpoints to enable access to around 20 billion resulting triples in a virtual integrated manner, i.e., the required data are transparently collected from different SPARQL endpoints. Proper load balancing among SPARQL endpoints is also ensured to reduce the query execution time. To this end, we have divided each tumour data into three categories, each of which is assigned a different colour – blue, pink and green – as shown in Figures 57 and 61. The green category contains only methylation results, pink contains expression exon results and all other data are grouped in the blue category. The ratio of the sizes is 1:3:4 for blue, pink, and green respectively.

In order to achieve proper load balancing, if we allocate one SPARQL endpoint to the blue category data (smallest) then we must assign three SPARQL endpoints to pink and four SPARQL endpoints to the green category data. We propose 17 SPARQL endpoints to be assigned for the complete TCGA level 3 data (around 33 tumours expected) distribution as shown in Figure 57. We assigned two SPARQL endpoints for blue, six endpoints for pink and nine endpoints for green category data.

Data are also balanced across each of the coloured category SPARQL endpoints according to cancer type (tumour). For example, in blue category, tumours 1-16 are stored in the first blue SPARQL endpoint and the remaining tumours (17-33) are stored in the second blue SPARQL endpoint. It is important to note that we have RDFized 27 tumours while in our data distribution diagram we show 33 tumours.

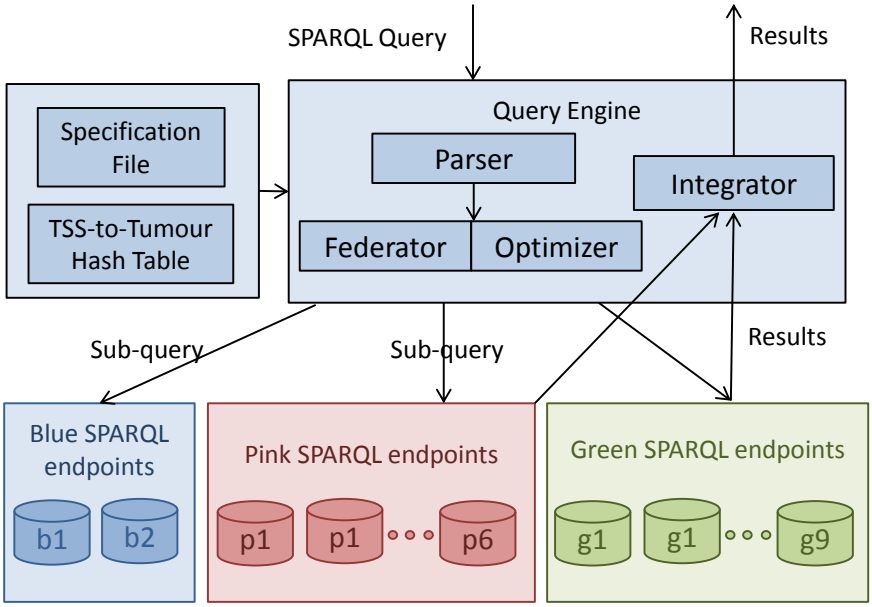


Figure 61: TopFed federated query processing model. TCGA tailored federated query processing diagram, showing system components.

This is because we are expecting around 33 cancer tumours[24] data to be made available by the TCGA data portal in the future. To achieve a similar size-oriented division, each of the SPARQL endpoints in the pink category contains either five or six tumours data as shown in Figure 57 and each of the first six SPARQL endpoints in the green category contain data for four tumours and each of the remaining three SPARQL endpoints contain three tumours data. Each of the three categories is used to create a conditional statement (labelled C-1, C-2, and C-3 given in Listing 14), used by the federated engine for source selection. For source selection, the predicates sets shown in Figure 57 (D, C, B, M, F, E, A and G) are also relevant. We further explain the decision model in Source Selection sub-section.

```
C–1 = {{p ∈ {D ∪ A ∪ G} ∨ {p = rdf:type ∧ o ∈ C}} ∧ {{S–Join
    (p, D ∪ C) ∨ P–Join(p, D ∪ C)} ∨ {!S–Join(p, M ∪ B ∪ E ∪
    F) ∧ !P–Join(p, M ∪ B ∪ E ∪ F)}}}

C–2 = {{p ∈ {E ∪ A ∪ G} ∨ {p = rdf:type ∧ o ∈ F}} ∧ {{S–Join
    (p, E ∪ F) ∨ P–Join(p, E ∪ F)} ∨ {!S–Join(p, M ∪ B ∪ D ∪
    C) ∧ !P–Join(p, M ∪ B ∪ D ∪ C)}}}

C–3 = {{p ∈ {M ∪ A} ∨ {p = rdf:type ∧ o ∈ B}} ∧ {{S–Join(p, M
     ∪ B) ∨ P–Join(p, M ∪ B)} ∨ {!S–Join(p, E ∪ F ∪ D ∪ C) ∧
    !P–Join(p, E ∪ F ∪ D ∪ C)}}}
```

Listing 14: Conditions for colour category selection

24 https://tcga-data.nci.nih.gov/tcga/

8.2.5 *TopFed federated query processing approach*

Before going into the details of our federated query processing model shown in Figure 61, we first briefly explain TopFed's index which comprise of an N3 specification file and a Tissue Source Site to Tumour (TSS-to-Tumour) hash table. The N3 specification file, shown in Listing 15, is devised based on the data distribution described in previous section. It contains metadata relevant for data distribution across SPARQL endpoints. For each SPARQL endpoint, its colour category, endpoint url, and the list of tumours data stored therein are specified. Moreover, the specification file also contains the various sets of predicates. In addition, we also create a Tissue Source Site to Tumour (TSS-to-Tumour[25]) hash table that contains key value pairs for TSS to tumour name. The TSS is the location identifier from where the results of the different tissues are obtained. This hash table was formed using "File_Sample_Map" files (containing file to patient barcode entries) provided as meta data, with every TCGA archive download via its Data Matrix portal[26]. This meta file provides a list of patient barcodes belonging to a particular cancer tumour. We extract the TSS part of patient barcode[27] and use this along with tumour name as a hash entry. Both N3 specification file and TSS-to-Tumour hash table are used by our federated query processor for efficient relevant data source (SPARQL endpoints) selection, which is explained in the next sub-section.

Given a SPARQL query, it is first parsed and then sent to the federator that makes use of the N3 specification file along with the TSS-to-Tumour hash table, in order to find the relevant sources for each of the triple pattern using Algorithm 8. The optimizer makes use of the source selection to generate an optimized sub-query execution plan. The optimized sub-queries are then forwarded to the relevant SPARQL endpoints. The results of each sub-query execution are integrated and the final query result set is generated.

8.2.6 *Source selection*

The goal of the source selection is to find the optimal list of relevant sources (i.e., SPARQL endpoints) against individual query triple pattern. According to the distribution of Figure 57, if we can infer the category colour and tumour number for a triple pattern then we only need to query a single endpoint for that triple pattern. For example, starting from the root node of Figure 57, we can go to the second level of the tree by knowing the category colour (blue, pink, and green). Further, at second level, if we know the tumour number then we can

25 https://topfed.googlecode.com/files/TSS-to-Tumour_hash_table.txt
26 TCGA Data Matrix: https://tcga-data.nci.nih.gov/tcga/dataAccessMatrix.htm
27 Patient barcode format: https://wiki.nci.nih.gov/display/TCGA/TCGA+Barcode

Algorithm 8 triple pattern source selection

Require: D_{blue} = {b_1, b_2}; D_{pink} ={p_1, p_2, ... p_6}; D_{green} ={g_1, g_2,... g_9}; T = {$t_1, t_2, ...t_n$}; tumourNo //data sources, query triple patterns, tumour number (can be null)

1: **for** each bgp ∈ T **do**
2: **for** each t_i ∈ bgp **do**
3: sources = null; c_1Sources = null; c_2Sources = null; c_3Sources = null; type = null; s = subj(t_i); p = pred(t_i); o = obj(t_i)
4: **if** bound(s) **then**
5: catColour = s.getCategorycolour() //get category colour from subject
6: tNo = s.getTumour() //get tumour from subject
7: **if** catColour = 'blue' **then**
8: sources = D_{blue}
9: **else if** catColour = 'pink' **then**
10: sources = D_{pink}
11: **else if** catColour = 'green' **then**
12: sources = D_{green}
13: **end if**
14: S_i = sources.filter(tNo) //this will return a single capable source
15: **else if** bound(p) **then**
16: **if** C-1 **then**
17: c_1Sources = D_{blue}
18: **end if**
19: **if** C-2 **then**
20: c_2Sources = D_{pink}
21: **end if**
22: **if** C-3 **then**
23: c_3Sources = D_{green}
24: **end if**
25: sources = c_1Sources ∪ c_2Sources ∪ c_3Sources
26: **if** sources = null **then**
27: sources = D_{blue} //only check for clinical properties
28: **end if**
29: **if** tumourNo ≠ null **then**
30: S_i = sources.filter(tumourNo)
31: **else**
32: S_i = sources
33: **end if**
34: **else if** !bound(p) ∧ !bound(s) **then**
35: // prune selected sources with ASK queries
36: **for** each s_i ∈ {D_{blue} ∪ D_{pink} ∪ D_{green}} **do**
37: **if** ASK(s_i, t_i) = true **then**
38: S_i = S_i ∪ {s_i}
39: **end if**
40: **end for**
41: **end if**
42: return S_i //reutrn the set of relevant sources for triple pattern t_i
43: **end for**
44: **end for**

```
@prefix tcga:<http://tcga.deri.ie/schema/>.
<http://tcga.deri.ie/set/setA> tcga:setName "A";
                  tcga:setElements "result", "chromosome", "
                      bcr_patient_barcode".
<http://tcga.deri.ie/set/setE> tcga:setName "E";
                  tcga:setElements "RPKM".
<http://tcga.deri.ie/set/setG> tcga:setName "G";
                  tcga:setElements "start", "stop".
<http://tcga.deri.ie/set/setM> tcga:setName "M";
                  tcga:setElements "position", "beta_value".
<http://tcga.deri.ie/endpoint/blue1>  tcga:category "blue";
                  tcga:endpointUrl "http://10.196.2.214:8890/
                      sparql";
                  tcga:containTumours "BLCA", "CESC", "HNSC",
                      "KIRP", "LAML".
<http://tcga.deri.ie/endpoint/blue2> tcga:category "blue";
                  tcga:endpointUrl "http://10.196.2.123:8890/
                      sparql";
                  tcga:containTumours "LGG", "LUSC", "PRAD", "
                      READ", "SKCM".
<http://tcga.deri.ie/endpoint/pink1> tcga:category "pink";
                  tcga:endpointUrl "http://10.196.2.130:8890/
                      sparql";
                  tcga:containTumours "BLCA", "CESC", "HNSC".
```

Listing 15: Part of the N3 specification file

reach to a single SPARQL endpoint to query. For each query triple pattern, our source selection algorithm tries to get such information using the specification file and type (star, path) of the join between the query triple patterns.

A star join between two triple patterns is formed if both of the triple patterns share the same subject. Consider the query given in Listing 16: the first two triple patterns form a star join and the last four triple patterns form a second star join. A path join between two triple patterns is formed if object of the first triple pattern is used as subject of the second triple pattern. For example, the second triple pattern form a path join with the third triple pattern in the query shown in Listing 16. Moreover, every TCGA patient is uniquely identified by its barcode of the format <TCGA-TSS-PatientNo>. For example, the patient barcode used in the first triple pattern of the Listing 16 query has a TSS identifier 18 and patient number 3406. This means we can infer tumour name/number from patient barcode using the TSS to tumour hash table.

As discussed in the Data distribution section, we have categorized all SPARQL endpoints into three different category colours named blue, pink, and green. Our source selection algorithm (cf. Algorithm 8) requires the set of SPARQL endpoints in each of the colour category and stores three different sets named D_{blue}, D_{pink}, and D_{green}. Moreover, it requires the tumour number *tumourNo*, which can be null and is obtained from the query as follow: if a triple pattern with

```
PREFIX tcga: <http://tcga.deri.ie/schema/>.
SELECT ?recordNo ?chromosom ?start ?stop ?mean
WHERE
{
?s       tcga:bcr_patient_barcode "TCGA-18-3406".
?s       tcga:result     ?recordNo.
?recordNo tcga:chromosome ?chromosom.
?recordNo tcga:start       ?start.
?recordNo tcga:stop        ?stop.
?recordNo tcga:seq_mean   ?mean.
}
```

Listing 16: TCGA query with bound predicate

```
PREFIX tcga: <http://tcga.deri.ie/schema/>
SELECT ?recordNo  ?start ?stop ?rpkm
WHERE
{
<http://tcga.deri.ie/TCGA-18-3406-e266> tcga:start    ?start.
<http://tcga.deri.ie/TCGA-18-3406-e266> tcga:stop    ?stop.
<http://tcga.deri.ie/TCGA-18-3406-e266> tcga:RPKM    ?rpkm.
}
```

Listing 17: TCGA query with bound subject

predicate tcga:bcr_patient_barcode and bound object containing the patient barcode form a star join with a triple pattern having predicate tcga:result, then by using the patient barcode value specified in the former triple pattern can be used to get the required tumour number using TSS-to-Tumour hash table. Our source selection algorithm runs for each basic graph pattern (BGP[28]) and for each individual triple pattern of BGP as follow.

If subject of the triple pattern is bound then we can get both the category colour and tumour name from the subject URI. The format of the TCGA URI is <http://tcga.deri.ie/Patient_barcode-ResultType>. The tumour name can be obtained from Patient_Barcode and the category colour can be inferred from ResultType. For example, if the first character is *e* (shortcut for exon-expression), then it belongs to the pink category. However, if the first character is *d* (shortcut for dna-methylation), then it belongs to the green category and all other characters belong to the blue category. Consider the query given in Listing 17: the tumour name can be obtained using hash table lookup for TSS 18 and the colour category is pink.

Source selection for a triple pattern with only bound predicate is more challenging. We have divided various predicates and classes of the TCGA data into different sets that are shown in Listing 18. Set *D* contains all the predicates that uniquely identify the blue category and set *C* contains a list of classes specific to it. The sets *B* and *M* uniquely identify the methylation, i.e., the green category while sets

28 http://www.w3.org/TR/sparql11-query/#BasicGraphPatterns

F and *E* are for the pink category. Sets *A* and *G* contain predicates that can be found in more than one colour category. Starting from the root of the source selection tree, if the condition *C-1* given in Listing 14 holds then all of the sources in blue category are relevant for that triple pattern. This means that if predicate *p* of the triple pattern is set member of {D ∪ A ∪ G} or it is equal to rdf:type and the object *o* belongs to set *C* and either the star or path join between *p* and {D ∪ C} is true or the star and path join of *p* with {M ∪ B ∪ E ∪ F} is false, then all of the sources in the blue category are relevant.

```
D = {seq_mean, reads_per_million_mirna_mapped, scaled_estimate
    , protein_expression_value}
C = {copy_number_result, snp_result, expression_gene_result,
    expression_protein_result, mirna_result, Clinical}
B = {dna_methylation_result}
M = {beta_value, position}
F = {expression_exon_result}
E = {RPKM}
A = {chromosome, result, bcr_patient_barcode}
G = {start, stop}
```

Listing 18: Predicate and class sets

Consider the third triple pattern of the query given in Listing 16: the predicate chromosome is set member of *A*, which means this predicate can be found in all of the endpoints. However, chromosome has a star join with seq_mean, which is unique for the blue category sources. Therefore, instead of selecting all of the sources (overestimated as in FedX, SPLENDID etc.), TopFed will only select D_{blue} as relevant sources that can be further filtered, provided that the tumourNo given as input to Algorithm 8 is not null. Similarly, *C-2* holds for D_{pink} and *C-3* holds for D_{green} relevant source selection. It is important to note that more than one condition (C-1, C-2, C-3) can be true for a triple pattern, therefore we check each of the three conditions individually and make a union of the sources as given in line 24 of Algorithm 8. Further, if none of the condition is true then we need to query the blue category sources because we did not list many of the blue category predicates as they are numerous.

For a triple pattern with bound object, we send SPARQL ASK queries including the triple pattern to all of the sources and select sources that pass the test. This is similar to the source selection technique used in FedX for all the triple patterns. Along with Algorithm 8, Figure 57 also provides a visual demonstration of our triple pattern-wise source selection.

As an example, consider the query of Listing 16 and the data distribution given in Figure 57. TopFed selects one source for the first triple pattern because we can obtain tumour number from the given patient barcode and this triple pattern only passes *C-1*. FedX selects three sources since every patient data can be found in each of the three colour categories exactly at one SPARQL endpoint. For the sec-

ond triple pattern, TopFed again selects only one source because *C-1* only holds. However, FedX selects all of the 17 sources as predicate tcga:result can be found in all of the endpoints. For each of the remaining triple patterns (3 to 6), TopFex selects only one source as tcga:seq_mean is unique for the blue category endpoints and the others triple patterns (3 to 5) has star join with it. We have only two endpoints in blue category, which is filtered to one using the tumour number given in triple pattern 1. FedX selects all of the 17 sources for tcga:chromosome, eight sources each for tcga:start, tcga:stop, and two sources for last triple pattern. In total, TopFed selects only six sources while FedX selects 52 to answer this query. Additionally, FedX also needs to send 102 (6*17) SPARQL ASK queries. We want to emphasize that we have replaced only source selection algorithm of FedX. The join order optimization and the join implementation remains the same.

8.3 RESULTS AND DISCUSSION

8.3.1 *Evaluation*

The goal of this evaluation is to support the claim that TopFed selects a significantly smaller number of sources for the same recall as FedX, thus achieving a good query execution performance for large datasets. We compare TopFed with the state-of-the-art approach for query federation (FedX) both in terms of the total number of sources selected and the execution time to achieve a 100% recall, using 10 TCGA benchmark SPARQL queries[29] of different shapes (i.e. star, path, and hybrid). A textual description of all the benchmark queries is given in Table 31. FedX has been shown previously [29; 82] to be the fastest and more precise SPARQL federated query engine (to the best of our knowledge). Therefore, we evaluate TopFed's query performance by comparing it with FedX.

8.3.1.1 *TCGA benchmark setup*

TCGA benchmark data consists of genomic results from 25 patients randomly selected from ten different tumour types and distributed across ten local SPARQL endpoints with the specifications given in Table 32. Furthermore, the benchmark N3 specification[30] file (used in the current experiments) assigns two, three, five SPARQL endpoints to the blue, pink, and green categories respectively.

We have selected ten SPARQL queries based on expert opinion reflecting typical requests on TCGA data. Further, we have categorized our benchmark queries into four different quadrants as shown in Ta-

29 Benchmark queries: http://goo.gl/UxUEXk
30 TopFed index: https://topfed.googlecode.com/files/loadDistribution.n3

Table 31: **Benchmark queries descriptions**

Query	Description
Q1	Get the chromosome, start, stop and mean copy number values of the patient TCGA-18-4721 for genome locations 554268 to 5994290
Q2	Get the chromosome, start, stop and mean exon-expression values of all the TCGA patients
Q3	Get the chromosome, position and mean methylation values of all the TCGA patients
Q4	Get the chromosome, start and stop values of the TCGA patient TCGA-C4-A0F6
Q5	Get the chromosome, start, stop values of all the TCGA patients
Q6	Get the chromosome, start, stop and miRNA values of the 20th record of TCGA patient TCGA-AB-2821
Q7	Get the chromosome, start and stop values of the TCGA patient TCGA-AB-2823 for mean sequence value of 0.0839
Q8	Get the chromosome, start, stop, mean protein expression and mean exon-expression values of the TCGA patient TCGA-18-3410
Q9	Get the chromosome, mean gene expression and mean methylation values of the TCGA patient TCGA-C5-A1BF
Q10	Get the chromosome, mean gene expression, mean exon expression and mean methylation values of all the TCGA patients

The corresponding SPARQL queries can be downloaded from http://goo.gl/UxUEXk.

Table 32: **Benchmark SPARQL endpoints specifications**

SPARQL endpoint	CPU	RAM	Hard Disk
virtuoso-blue1	2.2 GHz, i3	4 GB	300 GB
virtuoso-blue2	2.6 GHz, i5	4 GB	150 GB
virtuoso-pink1	2.53 GHz, i5	4 GB	300 GB
virtuoso-pink2	2.3 GHz, i5	4 GB	500 GB
virtuoso-pink3	2.53 GHz, i5	4 GB	300 GB
virtuoso-green1	2.9 GHz, i7	16 GB	256 GB SSD
virtuoso-green2	2.9 GHz, i7	8 GB	450 GB
virtuoso-green3	2.6 GHz, i5	8 GB	400 GB
virtuoso-green4	2.6 GHz, i5	8 GB	400 GB
virtuoso-green5	2.9 GHz, i7	16 GB	500 GB

Table 33: **Benchmark queries distribution**

	Single Colour	Cross-Colour
Star	2	2
Hybrid (star + path)	2	4

ble 33. A single colour query collects results from SPARQL endpoints listed in one of the three colour categories. A cross-colour query targets more than one colour category results. A hybrid query contains both star and path joins between various triple patterns. Moreover, we can also obtain the tumour number (to be used as input to Algorithm 8) from all of the hybrid queries. All of the benchmark data, including benchmark queries, can be found at the project website.

8.3.1.2 *Experimental results*

In order to show the effects of source selection on performance (runtime + recall of sources selected), the number of sources selected for each triple pattern of the query are added (equation 7). Let m_i equal the number of sources capable of answering a triple pattern t_i and S is the total number of available sources (10 in our benchmark). Then, for a query q with triple patterns $\{t_1, t_2, \ldots, t_n\}$, the total number of sources selected (triple pattern-wise sources selected) is given in equation 7.

$$\text{total number of sources selected} = \sum_{t=1}^{n} m_t : 0 \leqslant m_t \leqslant S \qquad (7)$$

The source selection results are shown in Figure 62. Overall, our source selection algorithm selects on average less than half of the sources selected by FedX. This is due to the possible overestimation of the sources by FedX while using SPARQL ASK queries for relevant source selection [95]. For example, any data source will likely match a triple pattern (?s, rdf:type, ?o). However, the same sources might not lead to any results at all once the actual mappings for ?s and ?o are included in a join evaluation. On the contrary, our source selection algorithm was designed to resolve the join types between query triple patterns specifically to avoid such overestimation (which can later greatly increase the query processing time as reflected in Table 34). Only in queries 5 and 10, TopFed selected sources are equal to FedX. The explanation for this can be found in the amount of useful information available in each query - both query 5 and query 10 are generic queries from which a tumour or a performance-improving colour category cannot be derived, because all logic conditions are exactly satisfied. Overall, TopFed selects the optimal (the actual required sources) number of sources with 100% recall for all of the benchmark queries.

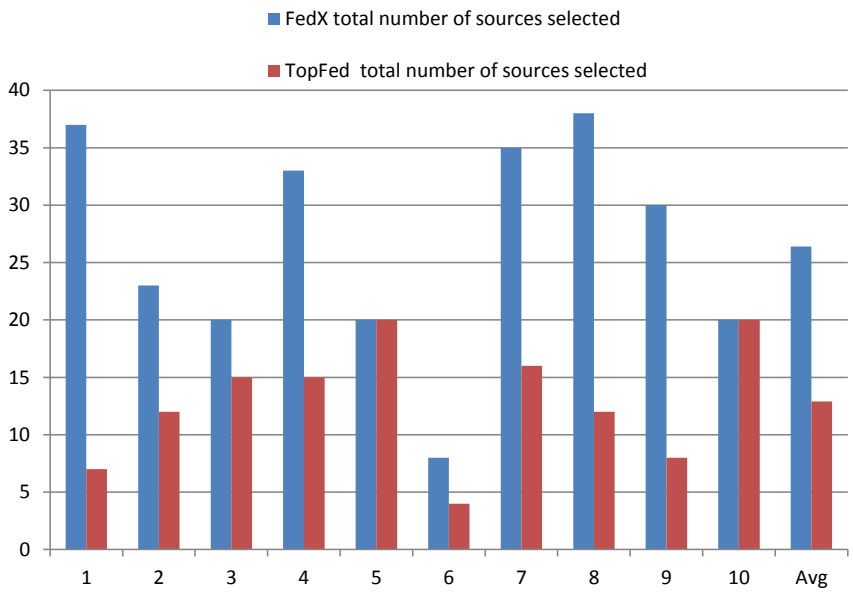


Figure 62: Efficient source selection. Comparison of the TopFed and FedX source selection in terms of the total number of triple pattern-wise sources selected. Y-axis shows the total triple pattern-wise sources selected for each of the benchmark query given in X-axis.

Table 34: **sectionage execution time for each query (based on a sample of 10)**

	FedX(first run)	FedX(cached)		TopFed	
Query No	**Execution Time(msec)**	**Execution Time(msec)**	**S.E**	**Execution Time(msec)**	**S.E**
1	913	401.2	5.22	341.5*	5.60
2	81619	81170.7	655.93	866.5*	22.08
3	82271	81817.8	653.22	666*	27.12
4	1199	367.6	6.88	262.7*	7.35
5	80423	78723.5	459.43	78691.5	458.70
6	837	416.9	8.38	246.1*	3.56
7	921	399.6	4.41	248.1*	7.20
8	900	89	2.45	72.7*	1.52
9	950.3	76.8	2.16	63.3*	1.89
10	912	63.6	1.99	49.6*	1.02
Average	25094.53	24352.67	180.01	8150.8	53.60

*Significant improvement.

Table 35: **Comparison of source selection average execution time (based on a sampling of 10)**

	FedX(first run)	FedX(cached)		TopFed	
Query No	Execution Time(msec)	Execution Time(msec)	S.E	Execution Time(msec)	S.E
1	530	11.7	0.35	28.1	0.98
2	487	11.4	0.67	5.2	0.57
3	470	11.9	0.78	5	0.42
4	510	12	0.52	23.6	1.57
5	473	9.8	0.65	4.8	0.29
6	371	9.9	0.38	21.7	0.68
7	521	10	0.39	24.4	0.76
8	483	9.5	0.45	29.5	0.86
9	496	9.8	0.39	20.1	0.99
10	456	10.6	0.40	7.4	0.58
Average	479.7	10.66	0.50	16.98	0.77

We have performed a two-tailed heteroscedastic t-test based on a sample of 10 (each query was run 10 times) to compare the source selection execution time. The source selection execution time and the standard error (S.E) obtained are presented in Table 35. On average, our source selection algorithm only requires 17 msec per query. This is because our N3 specification file is much smaller (only 43 lines) and we have created an in-memory Sesame repository to load and access this file. For the first run, the FedX source selection execution time is much higher. This delay is caused by the query engine sending a SPARQL ASK query for each of the query triple patterns, and for each of the sources. As explained above, FedX needs to issue 102 SPARQL ASK queries to perform source selection for the query in Listing 16 and the data distribution in Figure 57. In order to minimise the number of SPARQL ASK queries, FedX makes use of the cache to store the result of the recent SPARQL ASK request. Every time a query is issued, the engine first looks for a cache hit before issuing the actual SPARQL ASK query. To show the effect of the cache, we have rerun the same query 10 times after the first run and we have noticed a reasonable improvement. For a complete cached entries (100% cache hit), our source selection execution time is still comparable with FedX. It is important to note that all queries that are not specific to a patient (i.e, queries 2, 3, 5, 10), the TopFed source selection time is small (less than 10 msec). The reason is that the tumour number cannot be inferred from these queries and as a result less computation (index lookups) is required in the source selection Algorithm 8.

In Table 34, we compare the execution time of TopFed and FedEx for all of the benchmark queries using a two-tailed heteroscedastic t-test based on a sample of 10. It is important to mention that the query execution time was measured when the first result was retrieved, i.e., we did not iterate over all results. As an overall performance evaluation, the query execution time of TopFed is about one third to that of FedX. Specifically, TopFed significantly outperforms FedX in bench-

mark queries 2 and 3 related to exon expression and methylation, respectively. These queries select the complete set of results for all of the 25 patients. TopFed is able to infer from the query that the category colour should be pink and green, respectively, and issue the complete query to only the endpoints in the corresponding colour categories. In contrast, FedX is not able to perform such pre-processing, hence issuing the query to all endpoints. As a result, it has to collect results from all of the endpoints in the blue, pink, and green categories when only one of the categories can produce results for each query. As an example of the FedX approach addressing query 2, the triple pattern (?recordNo, tcga:chromosome, ?chromosom) relies on retrieving the results from all of the endpoints in both the blue and green categories, only to return an empty set of results, after making a star join with the triple pattern (?recordNo, tcga:RPKM, ?RPKM). We expect that our approach will generally lead to much faster resolution for queries of this nature, where a large number of triples is retrieved for a specific colour category. This reflects the improvement that TopFed's engine is able to determine those queries that will return empty sets prior to requesting the data. Although the benchmark query 5 results in a very large set of triples, the execution time for both systems is almost the same. As pointed out above, the reason for this is that the query is too generic and it is impossible to infer the category colour or tumour number.

8.4 AVAILABILITY OF SUPPORTING DATA

The TCGA data is available under the original TCGA Data Use Certification Agreement[31] and TopFed source code along with utilities are available under GNU GPL v3 licence at the project home page https://code.google.com/p/topfed/.

31 https://tcga-data.nci.nih.gov/docs/TCGA_Data_Use_Certification.pdf

9

LARGERDFBENCH: LARGE SPARQL ENDPOINT FEDERATION BENCHMARK

This chapter is based on [79] and provides the details of LargeRDFBench, a benchmark for SPARQL endpoint federation. The importance of Linked Data management and SPARQL queries has led to the development of several benchmarks (e.g., [3; 17; 32; 63; 91; 93; 102]) that allow assessing the performance of SPARQL query processing systems. However, all of these benchmarks (except FedBench [91]) have focused on the problem of query evaluation over local, centralised repositories. Hence, these benchmarks do not consider federated queries over multiple interlinked datasets hosted by different SPARQL endpoints. Moreover, most of them either rely on synthetic data (e.g., [17; 32; 93]) or synthetic queries [3].

While synthetic benchmarks allow generating datasets of virtually any size, they often fail to reflect reality [25]. In particular, previous works [25] point out that artificial benchmarks are typically highly structured while real Linked Data sources are less structured. Moreover, the synthetic queries should reflect the characteristics of the real queries (i.e., they should show typical requests on the underlying datasets) [8; 68]. Thus, synthetic benchmark results are rarely sufficient to extrapolate the performance of federation engines when faced with real data and in real queries. A trend towards benchmarks with real data and real queries (e.g., FedBench [91], DBPSB [63], BioBenchmark [102]) has thus been pursued over the last years but has so far not been able to produce federated SPARQL query benchmarks that reflect the data volumes and query complexity that federated query engines already have to deal with on the Web of Data. While the trend towards using real data and real queries in benchmarks addresses the need to reflect reality, most of the current benchmarks for SPARQL query execution rely only on a single performance criterion, i.e., the query execution time. Thus, they fail to provide results that allow a more fine-grained evaluation of SPARQL query processing systems to detect the components of systems that need to be improved [62; 87]. For example, performance metrics such as the *completeness and correctness of result sets* and the *effectiveness of source selection* both in terms of *total number of data sources selected*, and the corresponding *source selection time* are not addressed in the existing benchmarks [62; 87].

In this chapter, we address these drawbacks by presenting LargeRDFBench. (1) LargeRDFBench is an open-source benchmark for SPARQL endpoint query federation. To the best of our knowledge, this is the first federated SPARQL query benchmark with real data

(from multiple interlinked datasets pertaining to different domains) to encompass more than 1 billion triples. (2) We provide the SPARQL 1.0 and SPARQL 1.1 versions of the queries that are mostly collected from domain experts. In particular, we provide three types of queries that allow evaluating different aspects of the scalability of the current query federation frameworks. Note that for a particular LargeRDFBench query, the SPARQL 1.0 and SPARQL 1.1 versions of the query retrieve exactly the same result set. (3) We evaluate state-of-the-art SPARQL endpoint federation systems by using LargeRDFBench against several metrics including the source selection time, number of sources selected, result set correctness and completeness, and the query runtime. This fine-grained evaluation allows us to pinpoint the restrictions of current SPARQL endpoint federation systems when faced with large datasets, large intermediate results and large result sets. (4) Furthermore, we show that the current ranking of these systems based on simple queries differs significantly from their ranking when on more complex queries.

The rest of this chapter is structured as follows: We begin by providing an overview of the main components of a SPARQL query federation benchmark (short: benchmark) and key features that need to be considered while designing such a benchmark. Thereafter, we give an overview of related work and point out the current drawbacks of existing benchmarks in more detail (Section 9.2). In Section 9.3, we describe LargeRDFBench. In particular, we present the datasets and queries contained in the benchmark as well as the metrics used for benchmarking with LargeRDFBench. An evaluation of state-of-the-art systems based on LargeRDFBench and the metrics presented in the prior section follows. The results are discussed and we finally conclude. The benchmark and further evaluation results can be found at benchmark homepage.[1]

9.1 BACKGROUND

This section explains the main components of the SPARQL query processing benchmarks and the key features of each of these components that should be considered during the benchmark creation. In general, a SPARQL query benchmark can be regarded as consisting of three main components: (1) a set of RDF datasets, (B) a set of SPARQL queries and (3) a set of performance metrics.

Datasets: The *datasets* used in the federated SPARQL benchmark should complement each other in terms of the total number of triples, number of classes, number of resources, number of properties, number of objects, average properties and instances per class, average in-degrees, outdegrees and their resource wise distributions [25]. Duan et al. [25] combines all of these datasets features into a single com-

1 LargeRDFBench http://LargeRDFBench.googlecode.com

posite metric called *structuredness or cohesion*. For a given dataset, the structuredness value ranges [0,1] with 0 means less structured and 1 means high structured dataset. A federated SPARQL query benchmark should comprise datasets of varying structuredness values.

SPARQL Queries: According to previous works [3; 30], a federated SPARQL query benchmark should vary the queries it contains w.r.t. the following *query characteristics*: number of triple patterns, number of join vertices, mean join vertex degree, number of sources span, query result set sizes, mean triple pattern selectivities (should be mean Filtered triple pattern selectivities if SPARQL `FILTER` clause is attached to the triple pattern), join vertex types ('star', 'path', 'hybrid', 'sink'), and SPARQL clauses used (e.g., `LIMIT`, `OPTIONAL`, `ORDER BY`, `DISTINCT`, `UNION`, `FILTER`, `REGEX`).

Performance Metrics: Previous works [62; 87] show that the *result set completeness and correctness*, the *total triple pattern-wise sources selected*, the *number of SPARQL ASK requests* used during source selection, the *source selection time*, and the *overall query execution time* are important metrics to be considered in SPARQL query federation benchmarks. We thus decided to implement these measures in LargeRDFBench. Note that LargeRDFBench is a benchmark for SPARQL endpoint federation, in contrast to Linked Data federation [87].

9.2 THE NEED OF MORE COMPREHENSIVE SPARQL FEDERATION BENCHMARK

A large number of benchmarks for comparing SPARQL query processing systems have been developed over the last decade. These include the Waterloo Stress Testing Benchmark (WSTB) [3], the Berlin SPARQL Benchmark (BSBM) [17], the Lehigh University Benchmark (LUBM) [32], the DBpedia Sparql Benchmark (DBPSB) [63], FedBench [91], SP^{2}Bench [93], and the BioBenchmark [102]. WSTB, BSBM, DBPSB, SP^{2}Bench, and BioBenchmark were designed with the main goal of evaluating query engines that access data kept in a single repository. They are used for the performance evaluation of different triple stores. LUBM was designed for comparing the performance of OWL reasoning engines. However, all of these benchmarks do not consider distributed data and federated SPARQL queries, thus they are not further considered in the discussion.

SPLODGE [30] is a heuristic for automatic generation of federated SPARQL queries which is limited to conjunctive BGPs. Non-conjunctive queries that make use of the SPARQL `UNION`, `OPTIONAL` clauses are not considered. Thus, the generated set of synthetic queries fails to reflect the characteristics of the real queries. For example, the DBpedia query log [68] shows that 20.87%, 30.02% of the real queries contains SPARQL `UNION` and `FILTER` clauses, respectively. However,

Table 36: Queries distribution with respect to different SPARQL clauses.

	SPARQL Clauses						
Benchmark	LIMIT	OPTIONAL	ORDER BY	DISTINCT	UNION	FILTER	REGEX
FedBench	0%	7.14%	0%	0%	21.42%	7.14%	0%
LargeRDFBench	12.5%	25%	9.3%	28.1%	18.75%	31.25%	3.12%

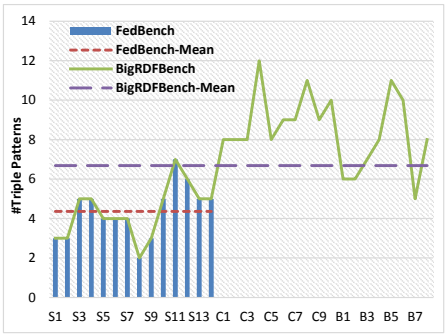
Table 37: Queries distribution with respect to join vertex types.

	Join Vertex Type			
Benchmark	Star	Path	Hybrid	Sink
FedBench	85.71%	57.14%	14.28%	35.71%
LargeRDFBench	75%	78.12%	40.62%	40.62%

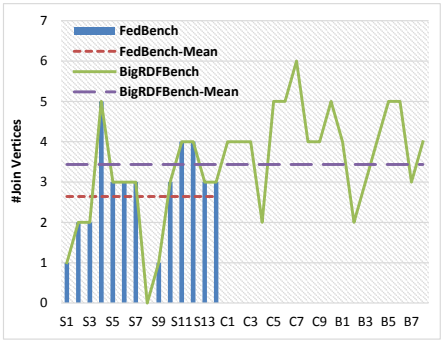
both of these clauses are not considered in SPLODGE queries generation. Moreover, the use of different SPARQL clauses and triple pattern join types greatly varies from one dataset to another dataset, thus making it almost impossible for automatic query generator to reflect the reality. For example, the DBpedia and Semantic Web Dog Food (SWDF) query log [8] shows that the use of the SPARQL `LIMIT` (27.99% for SWDF vs 1.04% for DBpedia) and `OPTIONAL` (0.41% for SWDF vs 16.61% for DBpedia) clauses greatly varies for these two datasets.

To the best of our knowledge, FedBench is the only benchmark that encompasses real-world datasets, commonly used federated SPARQL queries and a distributed data environment. It comprises a total of 14 queries for SPARQL endpoint federation and 11 queries for Linked Data federation approaches. In addition, this benchmark includes a dataset and queries from SP^2Bench. FedBench is commonly used in the evaluation of SPARQL query federation systems [94; 27; 61; 87; 77]. However, the real queries (excluding synthetic SP^2Bench benchmark queries) are low in complexity. The 11 Linked Data federation queries do not make use of any of the SPARQL clauses given in Table 36, the number of triple patterns included in the query ranges from 2 to 5, and the query result set sizes only ranges from 1 to 1216 (6/11 queries having result set size less than 51). As mentioned before, we are only interested in SPARQL endpoint federation queries. Therefore, the 14 SPARQL endpoint federation queries are further discussed in rest of the chapter.

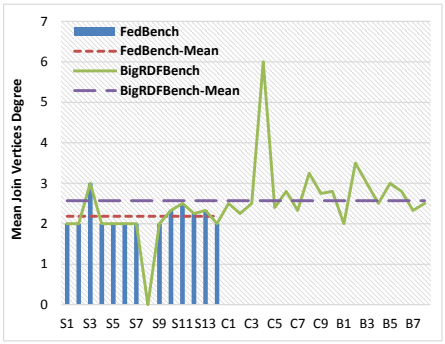
Table 36 and Figure 63 show that the FedBench SPARQL endpoint federation queries are also low in complexity and do not sufficiently complement (in terms of standard deviations for various query features discussed in previous section) each other's. Consequently, they may favour (in fact our evaluation given in Section 9.4.2.4 shows that indeed this is the case) a particular type of federation system. The number of Triple Patterns (#TP, ref. Figure 63a) included in the query ranges from 2 to 7. Consequently, the standard deviations of



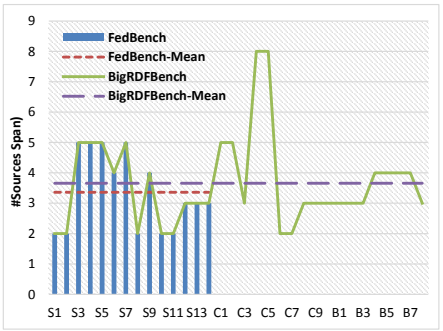
(a) #TP. (F.S.D. = ±1.33, B.S.D. = ±2.64)



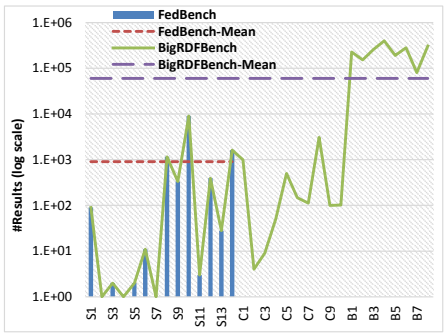
(b) #JV. (F.S.D. = ±1.33, B.S.D. = ±1.41)



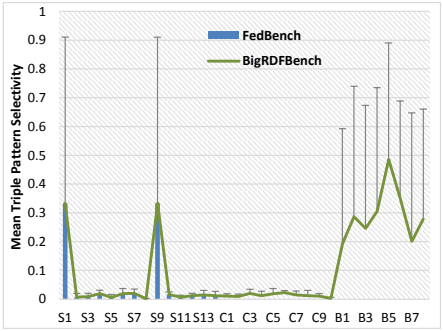
(c) MJVD. (F.S.D. = ±0.30, B.S.D. = ±0.76)



(d) #SS. (F.S.D. = ±1.27, B.S.D. = ±1.53)



(e) #R. (F.S.D. = ±2397, B.S.D. = ±113763)



(f) MTPS. Mean (F.S.D. = ±0.11, B.S.D. = ±0.14)

Figure 63: Comparison of query characteristics of FedBench and SlicedBench. **#TP** = Number of triple patterns, **#JV** = Number of join vertices, **MJVD** = Mean join vertices degree, **#SS** = Number of sources span, **#R** = Number of results, **MTPS** = Mean Triple Pattern Selectivity, **F.S.D.** = FedBench Standard Deviation, **B.S.D.** = LargeRDFBench Standard Deviation. X-axis shows the query name.

the number of Join Vertices (#JV, ref. Figure 63b), the Mean Join Vertices Degrees (#MJVD, ref. Figure 63c), and the number of Sources the query Span (#SS, ref. Figure 63d) are on the lower side. In particular, there are: 6/14 queries with #JV exactly equal to 3, 8/14 queries with #MJVD exactly equal to 2, and 5/14 queries with #SS exactly equal to 2. The query result set sizes (#R, ref. Figure 63e) are small (maximum 9054, 6/14 queries lead to a result set whose magnitude is less than 4). The query triple patterns are not highly selective in general (ref. Figure 63f). The important SPARQL clauses such `DISTINCT`, `ORDER BY` and `REGEX` are not used (ref. Table 36). Moreover, the SPARQL `OPTIONAL` and `FILTER` clauses are only used in a single query (i.e., LS7 of FedBench). Most importantly, the average query execution is small (about 2 seconds on average ref. Section 9.4.2.4). Finally, FedBench rely only on the number of endpoints requests and the query execution time as performance criteria. These limitations make it difficult to extrapolate how SPARQL query federation engines will perform when faced with the growing amount of data available on the Data Web based on FedBench results. Furthermore, a more fine-grained evaluation of the federation engines, to detect the components that need to be improved is not possible [62].

To address these limitations, we propose LargeRDFBench, a billion-triple benchmark which encompasses a total of 13 real, interconnected datasets of varying *structuredness* (ref. Figure 64) and real queries of varying complexities. (see Table 36 and Figure 63). Our benchmark includes all of the 14 SPARQL endpoint federation queries (which we named *simple queries*) from FedBench, as they are useful but not sufficient all alone. In addition, we provide 10 complex and 8 large data queries, which lead to larger result sets (see Figure 63e) and intermediary results (see triple pattern selectivities, Figure 63f) . Beside the central performance criterion, i.e., the query execution time, our benchmark includes result set completeness and correctness, effective source selection in terms of the total number of data sources selected, the total number of SPARQL ASK requests used and the corresponding source selection time. Our evaluation results (section 9.4.2) suggest that the performance of current SPARQL query federation systems on simple queries (i.e., FedBench queries) does not reflect the systems' performance on more complex queries. In addition, none of the state-of-the-art SPARQL query federation is able to fully answer the real use-case large data queries.

9.3 BENCHMARK DESCRIPTION

The idea behind this work was to design a benchmark based on real data and real queries that implements all of the key benchmark components features discussed in Section 9.1. The data was chosen to reflect the topology of the current Web of Data, with some of the

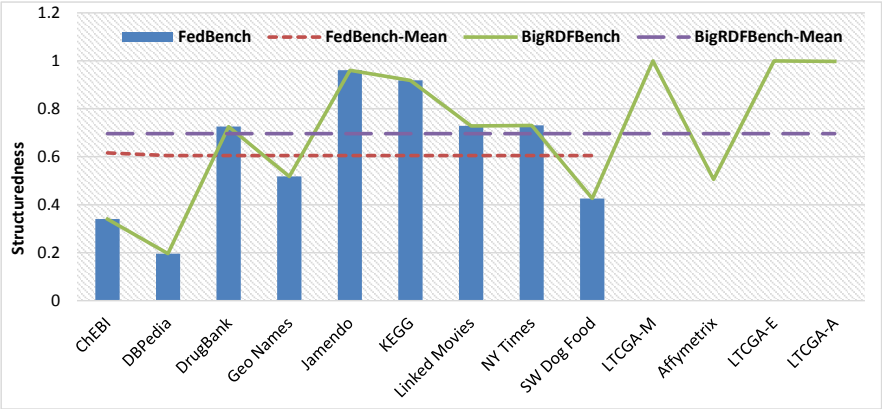


Figure 64: Structuredness. (FedBench Standard Deviation = ±0.26, LargeRDFBench Standard Deviation = ±0.28)

datasets being highly connected with other datasets while others are isolated (ref. Figure 65). Furthermore, some of the datasets are highly *structured* while others are low *structured* (ref. Figure 64). The queries were chosen to reflect a wide range of complexities w.r.t. the number of triple patterns they contain, the use of different SPARQL clauses, the triple patterns' selectivity, the number of join vertices, the mean join vertices degrees, the number of sources span, and the result set sizes they lead to (see Table 36 and Figure 63). The resulting benchmark, dubbed LargeRDFBench, consists consequently of three main components: (1) real-world datasets collected from different domains, (2) real queries mostly collected from domain experts and representing real use cases, and (3) a comprehensive set of fine-grained evaluation measures. In the following section, we present each of the three main components in detail.

9.3.1 *Benchmark Datasets*

Our benchmark consists of a total of 13 real-world datasets[2] of which 12 are interlinked. The datasets were collected from different domains as shown in Figure 65. We began by selecting all nine real-world datasets from Fedbench [91]. We added three sub-datasets from three different Linked TCGA live SPARQL endpoints [90] (i.e. Linked TCGA-A, Linked TCGA-M, and Linked TCGA-E) along with Affymetrix. We chose Linked TCGA because it is one of the first datasets that abides by many of the Vs of Big Data (Volume, Velocity, Value, ...) [89]. Moreover, Linked TCGA has a large number of links to Affymetrix, which we thus added to the list of our datasets. The addition of these four datasets enabled us to include real federated queries with large result set sizes (minimum 80459, see Figure 63e and benchmark homepage) into the benchmark. Figure 65 shows the topology of all 13 datasets in

2 Live SPARQL endpoints, datadumps URL's are given at project homepage: See footnote 1

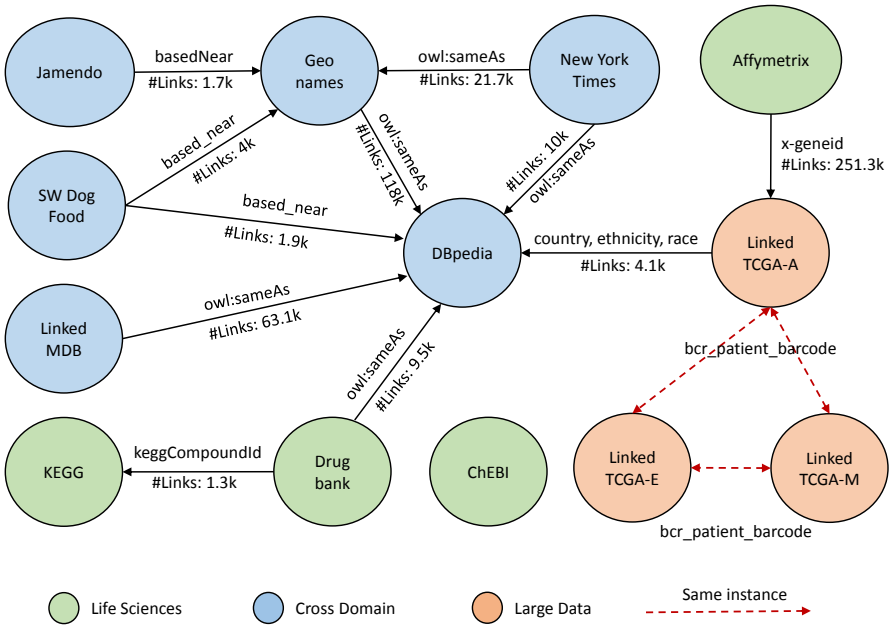


Figure 65: LargeRDFBench datasets connectivity diagram.

Table 38: LargeRDFBench datasets statistics. Structuredness is calculated according to [25] and is averaged in the last row.

Dataset	#Triples	#Subjects	#Predicates	#Objects	#Classes	#Links	Structuredness
Linked TCGA-M	415,030,327	83,006,609	6	166,106,744	1	-	1
Linked TCGA-E	344,576,146	57,429,904	7	84,403,422	1	-	1
Linked TCGA-A	35,329,868	5,782,962	383	8,329,393	23	251.3k	0.998
ChEBI	4,772,706	50,477	28	772,138	1	-	0.340
DBpedia-Subset	42,849,609	9,495,865	1,063	13,620,028	248	65.8k	0.196
DrugBank	517,023	19,693	119	276,142	8	9.5k	0.726
Geo Names	107,950,085	7,479,714	26	35,799,392	1	118k	0.518
Jamendo	1,049,647	335,925	26	440,686	11	1.7k	0.961
KEGG	1,090,830	34,260	21	939,258	4	30k	0.919
LinkedMDB	6,147,996	694,400	222	2,052,959	53	63.1k	0.729
New York Times	335,198	21,666	36	191,538	2	31.7k	0.731
Semantic Web Dog Food	103,595	11,974	118	37,547	103	1.6k	0.426
Affymetrix	44,207,146	1,421,763	105	13,240,270	3	246.3k	0.506
Total/Average	**1,003,960,176**	**165,785,212**	**2,160**	**326,209,517**	**459**	**818.7k**	**0.696**

LargeRDFBench while some other basic statistics like the total number of triples, the number of resources, predicates and objects, as well as the number of classes and the number of links can be found on project homepage. It is important to note that ChEBI has no link with any other benchmark dataset. However, its predicate "title" and DrugBank's predicate "genericName" display the same literal values. Similarly, the Linked TCGA-A predicate "drug_name" and DrugBank's "genericName" display the same values. Thus, they can be used in federated SPARQL queries. Furthermore, each Linked TCGA patient (uniquely identified by bcr_patient_barcode) expression data is distributed across the three large data datasets, i.e., Linked TCGA-A, Linked TCGA-E, and Linked TCGA-M (further explained in 9.3.1.3 subsection Large Data). Further datasets statistics can be found in Table 38.

The datasets in LargeRDFBench belong to three categories: Cross-domain, Life Sciences domain and large data.

9.3.1.1 *Cross-domain Datasets*

This category comprises datasets which pertain to several domains including news, movies, music, semantic web conferences and geography. These datasets include a 1) subset from DBpedia comprising infoboxes and the instance types, 2) the music knowledge base called Jamendo, 3) LinkedMDB , a knowledge base containing movie and actor information, 4) GeoNames , which contains geographical data about persons, locations, as well as organisations, 5) Semantic Web Dog Food which describes Semantic Web conferences and publications, and 6) a knowledge base containing news from the New York Times . Figure 65 shows the links between these data sources.

9.3.1.2 *Life Sciences Domain*

Life Sciences domain data sources include 1) Drugbank, a knowledge base containing information pertaining to drugs, their composition and their interactions with other drugs, 2) the Kyoto Encyclopedia of Genes and Genomes (KEGG) which contains further information about chemical compounds and reactions with a focus on information relevant for geneticists, 3) the Chemical Entities of Biological Interest (ChEBI) knowledge base which describes the life sciences domain from a chemical point of view and 4) the Affymetrix dataset that contains the probesets used in the Affymetrix microarrays.

9.3.1.3 *Large Data: Linked TCGA*

Linked TCGA is the RDF version of Cancer Genome Atlas[3] (TCGA) presented in [90]. This knowledge base contains cancer patient data generated by the TCGA pilot project, started in 2005 by the National Cancer Institute (NCI) and the National Human Genome Research Institute (NHGRI). Currently, Linked TCGA comprises a total of 20.4 billion triples[4] from 9000 cancer patients and 27 different tumour types. For each cancer patient, Linked TCGA contains expression results for the DNA methylation, Expression Exon, Expression Gene, miRNA, Copy Number Variance, Expression Protein, SNP, and the corresponding clinical data.

We selected the data of 306 patients distributed evenly across 3 different cancer types, i.e. Cervical (CESC), Lung squamous carcinoma (LUSC) and Cutaneous melanoma (SKCM). The selection of the patients was carried out by consulting domain experts. This data is hosted in three TCGA SPARQL endpoints with all DNA methylation

3 http://cancergenome.nih.gov/
4 http://tcga.deri.ie/

Table 39: LargeRDFBench query characteristics. (**#TP** = total number of triple patterns in a query, Query structure = **S**tar, **P**ath, **H**ybrid, **#Src** = number of sources span, **#Res.** = total number of results).

LargeRDFBench Queries									
Qry	#TP	Struct.	#Src	#Res.	Qry	#TP	Struct.	#Src	#Res.
Simple Queries					C3	8	H	3	9
S1	3	S	2	90	C4	12	S	8	50
S2	3	S	2	1	C5	8	H	8	500
S3	5	H	5	2	C6	9	H	2	148
S4	5	P	5	1	C7	9	H	2	112
S5	4	P	5	2	C8	11	H	3	3067
S6	4	P	4	11	C9	9	H	3	100
S7	4	P	5	1	C10	10	H	3	102
S8	2	-	2	1159	Large Data Queries				
S9	3	P	4	333	L1	6	P	3	227192
S10	5	H	2	9054	L2	6	H	3	152899
S11	7	H	2	3	L3	7	H	3	257158
S12	6	H	3	393	L4	8	H	4	397204
S13	5	H	3	28	L5	11	H	4	190575
S14	5	H	3	1620	L6	10	H	4	282154
Complex Queries					L7	5	H	4	80459
C1	8	H	5	1000	L8	8	H	3	306705
C2	8	H	5	4					

data in the first endpoint, all Expression Exon data in the second endpoint, and the remaining data in the third endpoint. Similarly, we created three different datasets namely Linked TCGA-M, Linked TCGA-E, and Linked TCGA-A containing methylation, exon, and all remaining data, respectively. Further statistics about these three datasets can be found at the project homepage.

9.3.2 *Benchmark Queries*

LargeRDFBench comprises a total of 32 queries for *SPARQL endpoint federation approaches*. These queries are divided into three different types: the 14 simple queries (namely S1-S14) are from FedBench). The 10 complex queries C1-C10 and the 8 large data dubbed C1-C8 were created by the authors with the help of domain experts. Table 39, Table 36 and Figure 63 shows key features of these queries. Further query statistics can be found on project home page.

```
SELECT ?party ?page WHERE {
dbpedia:Barack_Obama dbpedia:party ?party .
?x nyt:topicPage ?page .
?x owl:sameAs dbpedia:Barack_Obama . }
```

Listing 19: Return Barack Obama's party membership and news pages. Prefixes are ignored for simplicity

```
SELECT DISTINCT ?drug ?drugDesc ?molecularWeightAverage
    ?compound ?ReactionTitle ?ChemicalEquation WHERE {
 ?drug drugbank:description ?drugDesc .
 ?drug drugbank:drugType drugtype:smallMolecule .
 ?drug drugbank:keggCompoundId ?compound.
 ?enzyme kegg:xSubstrate ?compound .
 ?Chemicalreaction kegg:xEnzyme ?enzyme .
 ?Chemicalreaction kegg:equation ?ChemicalEquation .
 ?Chemicalreaction purl:title ?ReactionTitle
 OPTIONAL {
   ?drug drugbank:molecularWeightAverage ?
       molecularWeightAverage
   FILTER (?molecularWeightAverage > 114) } }
Limit 1000
```

Listing 20: Find the equations of chemical reactions and reaction title related to drugs with drug description and drug type 'smallMolecule'. Prefixes are ignored for simplicity

9.3.2.1 *Simple Queries*

In comparison to the other queries in the benchmark, the queries in this category comprise the smallest number of triple patterns, which ranges from 2 to 7. These queries require retrieving data from 2 to 5 data sources (ref. Figure 63d). Moreover, they only use a subset of the SPARQL clauses as shown in Table 36(see FedBench row as all of the simple queries are from FedBench). Amongst others, they do not use `LIMIT`, `REGEX`, `DISTINCT` and `ORDER BY` clauses. Finally, we will see in the evaluation section that the query execution time for such queries are small (around 2 seconds for FedX).

An example of such query is shown in Listing 19. It is important to mention that we removed the `FILTER (?mass > '5')` from the FedBench life sciences query LS7 (S14 in LargeRDFBench) because the KEGG drug mass is a string. Thus, using this operator on KEGG would lead to semantics different from that intended in the original query. Consequently, the result set size changes from 114 to 1620 rows.

9.3.2.2 *Complex Queries*

The complex queries were defined to address the restrictions of simple queries with respect to the number of triple patterns they use,

```
SELECT ?methylationCNTNAP2 WHERE {
 ?s affymetrix:x-symbol bio2rdfSymbol:CNTNAP2.
 ?s affymetrix:x-geneid ?geneId.
 ?geneId rdf:type tcga:expression_gene_lookup.
 ?geneId tcga:chromosome ?lookupChromosome.
 ?geneId tcga:start ?start.
 ?geneId tcga:stop  ?stop.
 ?uri tcga:bcr_patient_barcode ?patient .
 ?patient tcga:result ?recordNo .
 ?recordNo tcga:chromosome ?chromosome.
 ?recordNo tcga:position ?position.
 ?recordNo tcga:beta_value ?methylationCNTNAP2.
 FILTER (?position >= ?start && ?position <= ?stop &&
     str(?chromosome) = str(?lookupChromosome)) }
```

Listing 21: Get the methylation values for CNTNAP2 gene of all the cancer patients. Prefixes are ignored for simplicity

the SPARQL clauses, and the small query execution times. Consequently, queries in this category rely on at least 8 triple patterns. In addition, they were designed to use more SPARQL clauses, especially, `DISTINCT`, `LIMIT`, `FILTER` and `ORDER BY`. Later, we will see in the evaluation that the query execution time for complex queries reaches up to more than 10 minutes. An example of such query is shown in Listing 20.

9.3.2.3 *Large Data Queries*

The large data queries were designed to test the federation engines for real large data use cases, particularly in life sciences domain. These queries span over large data sets (such as Linked TCGA-E, Linked TCGA-M) and involve processing large intermediate result sets (usually in hundreds of thousands, see mean triple pattern selectivities in Figure 63f) or lead to large result sets (minimum 80459, see Figure 63e). In order to collect real queries with these characteristics, we contacted different domain experts and obtained a total of 8 large data queries to be included in our benchmark. An example of such query is given in Listing 21.

9.3.3 *Performance Metrics*

As discussed in Section 9.1, previous works [62; 87] suggest that the following five metrics are important to evaluate the performance of federation engines: (1) the total number of triple pattern-wise (TPW) source selected during the source selection, (2) the total number of SPARQL ASK requests submitted to perform (1), (3) the completeness (recall) and correctness (precision) of the query result set retrieved, (4) the average source selection time and (5) the average query execution time. The source index/data summaries generation time and index

compression ratio (i.e., index to dataset ratio) are also of central importance when evaluating endpoints. However, they are not applicable to index-free approaches such as FedX [94]. Previous works [87] show that an overestimation of triple pattern-wise sources selected can greatly increase the overall query execution time. This is because extra network traffic is generated and unnecessary intermediate results are retrieved, which are excluded after performing all the joins between query triple patterns. The time consumed by the SPARQL ASK queries during the source selection is directly added into the source selection time which in turns added into the overall query execution time.

9.4 EVALUATION

In this section, we evaluate state-of-the-art SPARQL query federation systems by using both SPARQL 1.0 and SPARQL 1.1 versions of LargeRDFBench queries. We first describe our experimental setup in detail. Then, we present our evaluation results. All data used in this evaluation can be found on the benchmark homepage.

9.4.1 *Experimental Setup*

Each of the 13 Virtuoso SPARQL endpoint used in our experiments was installed on a separate machine. The specification of each of the machines is given on the project home page. To avoid server bottlenecks, we started the two largest endpoints (i.e., Linked TCGA-E and Linked TCGA-M) in machines with high processing capabilities. All experiments (i.e., the federation engines themselves) were ran on a separate Linux machine with a 2.70GHz i7 processor, 8 GB RAM and 500 GB hard disk. We used the default Java Virtual Machine (JVM) initial memory allocation pool (Xms) size of 40MB and the maximum memory allocation pool (Xmx) size of 512MB. The experiments were carried out in a local network, so the network costs were negligible. Each query was executed 10 times and results were averaged. The query timeout was set to 20 min (1.2×10^6 ms) both for simple and complex queries and 1 hour (3.6×10^6 ms) for large data queries.

We compared five SPARQL endpoint federation engines[5] – FedX [94], SPLENDID [27], ANAPSID [27], FedX+HiBISCuS [87], SPLENDID+HiBISCuS [87] – on all of the 32 benchmark queries. Note that HiBISCuS [87] is only a source selection approach and FedX+HiBISCuS and SPLENDID+HiBISCuS are the HiBISCuS extensions of the FedX and SPLENDID query federation engines, respectively. To the best of our knowledge, the five systems we chose are the most state-of-the-art SPARQL endpoint federation engines [87]. Of all the systems,

5 Versions available as of October 2014

Table 40: Comparison of index construction time, compression ratio, and support for index update. (NA = Not Applicable).

	FedX	SPLENDID	ANAPSID	HiBISCuS
Index Gen. Time(min)	NA	190	6	92
Compression Ratio(%)	NA	99.998	99.999	99.998
Index update?	NA	✗	✗	✓

only ANAPSID and HiBISCuS perform *join-aware* Triple Pattern-Wise Source Selection (TPWSS).

9.4.2 *SPARQL 1.0 Experimental Results*

9.4.2.1 *Index Construction Time and Compression Ratio*

Table 40 shows a comparison of the index/data summaries construction time and the compression ratio[6] of the selected approaches. A high compression ratio is essential for fast index lookups during source selection and query planning. FedX does not rely on an index and makes use of a combination of SPARQL ASK queries and caching to perform the whole of the source selection steps it requires to answer a query. Therefore, these two metrics are not applicable for FedX. As pointed out in [87], ANAPSID only stores the set of distinct predicates corresponding to each data source. Therefore, its index generation time and compression ratio are better than that of HiBISCuS and SPLENDID on our benchmark.

9.4.2.2 *Efficiency of Source Selection*

We define efficient source selection in terms of: (1) the total number of triple pattern-wise sources selected (#TP), (2) the total number of SPARQL ASK requests (#AR) used to obtain (1), and (3) the source selection time (SST). Table 41 shows the results of these three metrics for the selected approaches.

Overall, ANAPSID is the most efficient approach in terms of total TPW sources selected, HiBISCuS is the most efficient in terms of total number of SPARQL ASK requests used, and FedX (100% cached) is the fastest in terms of source selection time (see Table 41). Still, FedX (100% cached) clearly overestimates the set of capable sources (474 in FedX vs. 229 optimal). FedX (100% cached) is clearly outperformed by ANAPSID (255 sources selected in total) and HiBISCuS (302 sources selected in total). FedX (100% cached)'s poorer performance is due to FedX only performing TPWSS while both HiBISCuS and ANAPSID perform *join-aware* TPWSS. As mentioned before, such overestimation of sources can be very costly because of the extra network traffic and irrelevant intermediate results retrieval. The effect

6 Compression ratio = 100*(1 - index size/total data dump size)

Table 41: Comparison of the source selection in terms of total triple pattern-wise sources selected **#TP**, total number of SPARQL ASK requests **#AR**, and source selection time **SST** in msec. **SST*** represents the source selection time for FedX (100% cached i.e. #A =0 for all queries). (**T/A** = Total/Avg., where Total is for #TP, #AR, and Avg. is SST, SST*)

	FedX				SPLENDID			ANAPSID			HiBISCuS			Optimal
Query	#TP	#AR	SST	SST*	#TP	#AR	SST	#TP	#AR	STT	#TP	#AR	SST	#TP
S1	15	39	238	**5**	15	34	622	**3**	**23**	227	4	26	322	3
S2	**3**	39	229	**6**	**3**	9	380	**3**	**1**	46	**3**	13	201	3
S3	12	65	275	**5**	12	2	358	**5**	2	70	**5**	**0**	52	5
S4	19	65	270	**7**	19	2	340	**5**	3	74	**5**	**0**	130	5
S5	11	52	268	**8**	11	1	330	**4**	1	65	**4**	**0**	90	4
S6	9	52	245	**5**	9	2	303	9	10	197	**8**	**0**	96	8
S7	13	52	248	**6**	13	2	354	**6**	5	273	**6**	**0**	149	6
S8	**1**	26	223	**5**	**1**	**0**	189	**1**	**0**	51	**1**	**0**	9	1
S9	15	39	240	**6**	15	34	592	15	23	356	**9**	**26**	449	3
S10	12	65	296	**5**	12	1	334	**5**	16	262	**5**	**0**	250	5
S11	7	91	300	**7**	7	2	299	**7**	0	333	**7**	**0**	12	7
S12	10	78	260	**5**	10	1	355	**7**	4	105	8	**0**	115	6
S13	9	65	262	**3**	9	2	262	**5**	24	180	7	**0**	132	5
S14	6	65	268	**5**	6	1	252	**5**	2	81	6	**0**	94	7
T/A	142	793	258	**5**	142	93	355	80	114	165	**78**	65	**150**	**67**
C1	11	104	308	**7**	11	1	291	**8**	1	72	9	**0**	120	8
C2	11	104	307	**6**	11	1	347	**8**	2	180	9	**0**	23	8
C3	21	104	318	**5**	21	3	350	**10**	33	549	11	**0**	230	10
C4	28	156	360	**7**	28	**0**	230	28	32	451	**18**	**0**	45	18
C5	33	104	315	**6**	33	**0**	199	**8**	3	156	10	**0**	56	8
C6	24	117	430	**5**	24	**0**	245	**9**	3	90	**9**	**0**	450	9
C7	17	117	436	**7**	17	2	422	**9**	5	380	**9**	**0**	168	9
C8	25	143	402	**4**	25	2	300	**11**	2	308	**11**	**0**	200	11
C9	16	117	400	**6**	16	2	480	**9**	16	185	**9**	**0**	180	9
C10	13	130	350	**8**	13	**0**	240	**11**	6	160	**11**	**0**	150	11
T/A	199	1196	363	**6**	199	11	310	111	103	253	**106**	**0**	**162**	**101**
L1	14	78	282	**5**	14	12	720	**6**	10	260	14	**0**	124	6
L2	10	78	279	**7**	10	1	230	**6**	5	142	10	**0**	94	6
L3	10	91	314	**9**	10	2	314	**7**	5	146	11	**0**	99	7
L4	18	104	321	**7**	18	**0**	198	**8**	8	338	16	**0**	80	8
L5	21	143	400	**5**	21	2	277	**12**	31	10255	20	**0**	130	11
L6	20	130	419	**4**	20	2	298	**10**	52	13173	18	**0**	160	10
L7	20	65	320	**6**	20	13	240	**6**	7	1822	9	**0**	270	5
L8	20	104	366	**7**	20	12	700	**9**	17	404	20	**0**	170	8
T/A	133	793	337	**6**	133	44	372	**64**	135	3317	118	**0**	**140**	**61**
Net T/A	474	2782	311	**6**	474	148	345	**255**	352	980	302	**65**	**151**	**229**

Table 42: Result set completeness and correctness: Systems with incomplete precision and recall. The values inside brackets show the LargeRDFBench query version, i.e., SPARQL 1.0 and SPARQL 1.1. For queries L2, L3, and L5 FedX and its HiBiSCUS extension produced zero results, thus both precision, recall is zero and F1 is undefined for these queries. (**NA** = SPARQL 1.1 not applicable, **TO** = Time out)

System	C7 (v1.0, v1.1)			L1 (v1.0, v1.1)			S14 (v1.1)			C6 (v1.1)		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
FedX	0.25	0.19	0.22	TO	TO	TO	1	0.65	0.78	1	0.97	0.98
FedX+HiBISCuS	0.25	0.19	0.22	TO	TO	TO	NA	NA	NA	NA	NA	NA
ANAPSID	1	1	1	1	0.14	0.25	1	1	1	1	1	1

of such overestimation is consequently even more critical while dealing with large data queries. HiBISCuS is better than ANAPSID in terms of total TPW source selected both for simple (78 for HiBISCuS and 80 for ANAPSID) and complex (106 for HiBISCuS and 111 for ANAPSID) queries. For large data queries (118 for HiBISCuS and 64 for ANAPSID), HiBISCuS is not able to skip many sources. This is because of the approach being based on making use of different URI authorities to perform source pruning [87]. However, most of the large data queries come from Linked TCGA with single URI authority (i.e., `tcga.deri.ie`). Hence, HiBISCuS tends to overestimate the number of sources in this case. On the other hand, ANAPSID makes use of SPARQL ASK requests combined with SSGM (Star Shaped Group Multiple Endpoints) [61] to skip a large number of sources. However, SPARQL ASK queries are expensive compare to local index lookups, as performed in HiBISCuS.

9.4.2.3 *Completeness and Correctness of Result Sets*

Two systems can only be compared to each other if they provide the same results for a given query execution. Table 42 shows the federation engines and the corresponding LargeRDFBench queries for which complete and correct results were not retrieved by at least one of the system. All those queries for which every system either timed out or resulted in runtime errors are not included, since results completeness and correctness cannot be determined in such cases. Here, SPLENDID and its HiBISCuS extension are the only systems that provide complete and correct results provided that the query is fully executed within the time out limit. The incomplete results generated by some of the systems can be due to a number of reasons, e.g., their join implementation, the type of network [62], the use of an outdated index or cache or even endpoints restrictions on the maximum result set sizes. However, in our evaluation we always used an up-to-date index and cache, there was no restriction on SPARQL endpoints maximum result set sizes, and a dedicated local network. Thus, the sole reason (to the best of our knowledge) for the systems at hand not providing complete/correct result is the type of the *join* used and

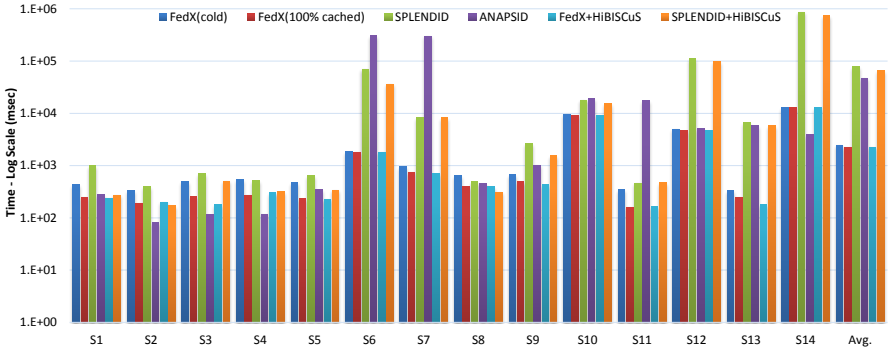


Figure 66: Query execution time for simple category queries.

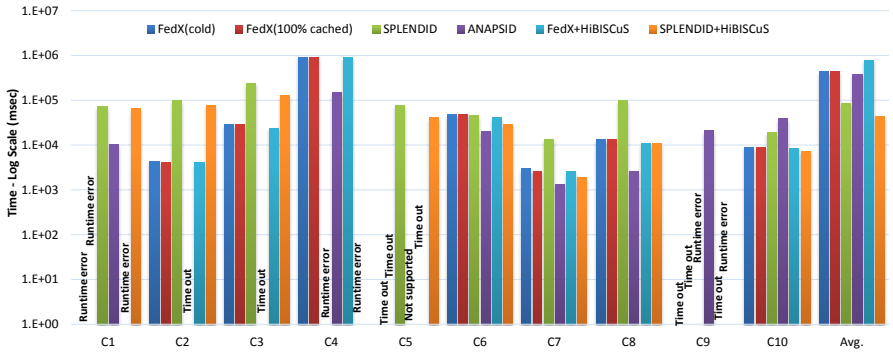


Figure 67: Query execution time for complex category queries.

its implementation. For query C7, FedX and its HiBISCuS extension were able to retrieve 85 records instead of the actual result set size, i.e., 112. For query L1, ANAPSID retrieved 35810 results while the actual result set size for this query is 227192. For the same query, both SPLENDID and FedX along with their extensions were only able to produce less than 30K results within timeout limit of one hour. For the SPARQL 1.1 versions of S14, FedX retrieved 1054 of the expected 1620 results while for C6 145 of the 148 results were retrieved.

9.4.2.4 *Query Execution Time*

The query execution time has often been used as key metric to compare federation engines. Figure 66, Figure 67, and Table 43 show the query execution time of the selected approaches for simple, complex, and large data queries, respectively. Note that we considered each time-out to be equal to a runtime of 20min while computing the average runtimes presented in Figure 66 and Figure 67. The query execution time was calculated once all the results were retrieved from the result set iterator. Overall, our results are rather surprising as no system is best over all query types.

- *Simple queries*: FedX+HiBISCuS and FedX clearly outperform the remaining systems (see Figure 66). In particular, FedX and its extension were better than SPLENDID+HiBISCuS in 12/14

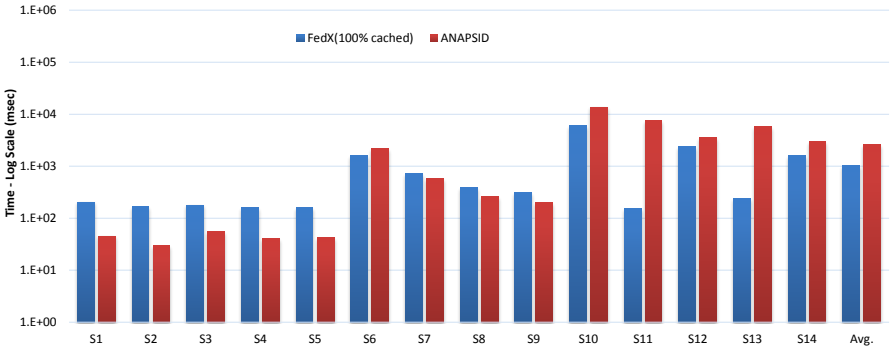


Figure 68: Query execution time for the SPARQL 1.1 version of the simple queries of LargeRDFBench.

queries. On the other hand, SPLENDID+HiBISCuS was better than ANAPSID in 8/14 queries, which in turn was better than SPLENDID in 10/14 queries.

- *Complex queries*: SPLENDID+HiBISCuS performed better than SPLENDID and was followed by ANAPSID, FedX+HiBISCuS and FedX. ANAPSID is better than SPLENDID+HiBISCuS in 4/7 comparable (those for which complete and correct results are retrieved by both systems) queries, SPLENDID+HiBISCuS is better than FedX and FedX+HiBISCuS in 5/7 comparable queries, which in turn better than SPLENDID in 5/7 comparable queries.

- *Large Data queries*: The most important result for large data queries is that no system can be regarded as superior because all systems are only able to produce complete results for a single query (i.e., L7). This shows the current implementation of query planning strategies (i.e., bushy trees in ANAPSID, left-deep trees in FedX, and dynamic programming [97] in SPLENDID) and join techniques (i.e., adaptive group and dependent join in ANAPSID, bind and nested loop in FedX, and bind, hash in SPLENDID) in the selected systems are not mature enough to deal with large data. For the only one comparable large data query, ANAPSID perform better than other systems. All of the errors thrown by the systems are available at benchmark homepage.

In a nutshell, our results clearly suggests that benchmarks with only simple queries with small number of result sets are not sufficient to make a fair judgment of the performance of the SPARQL query federation engines. The performance of these systems are greatly affected once the queries goes from small to complex and large data. Furthermore, the current state-of-the-art SPARQL query federation systems are not yet ready to deal with large data queries pertaining to real large data use cases.

Table 43: Runtimes on large data queries. **F(c)** = FedX (cold), **F(w)** = FedX(100% cached), **S** = SPLENDID, **A** = ANAPSID, **F+H** = FedX+HiBISCuS, **S+H** = SPLENDID+HiBISCuS. (**TO** = Time out after 1 hour, **ZR** = zero results, **IR** = incomplete results, **RE** = runtime error). Times are in seconds.

Query	F(c)	F(w)	S	A	F+H	S+H
L1	TO	TO	TO	IR	TO	TO
L2	ZR	ZR	TO	TO	ZR	TO
L3	ZR	ZR	TO	ZR	ZR	TO
L4	TO	TO	TO	TO	TO	TO
L5	ZR	ZR	TO	RE	ZR	TO
L6	TO	TO	TO	TO	TO	TO
L7	122	122	114	105	119	114
L8	TO	TO	TO	TO	TO	TO

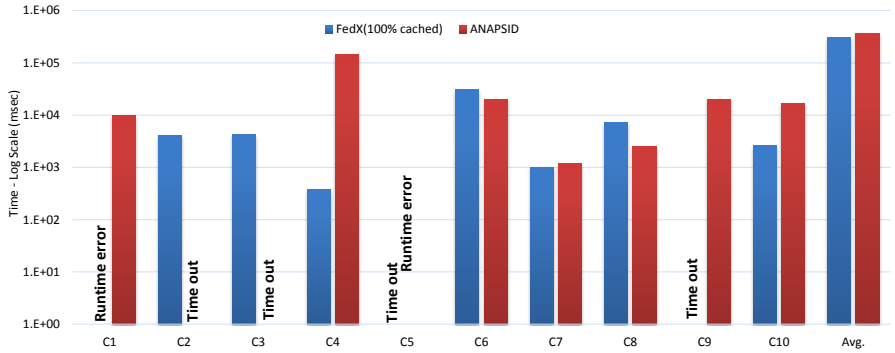


Figure 69: Query execution time for the SPARQL 1.1 version of the complex queries of LargeRDFBench.

9.4.3 *SPARQL 1.1 Experimental Results*

As per current implementations (October 2014), only ANAPSID and FedX support SPARQL 1.1 queries. Thus, they are the only frameworks we compared on the simple and complex SPARQL1.1 queries. For large data queries, the results remained the comparable to those presented before. Note that our SPARQL 1.1 version of the queries make use of the SPARQL `SERVICE` clause, which means the TPWSS is already performed. Furthermore, it is optimally chosen by manually looking at the intermediate results from all the data sources for a given query. Thus, the results presented in Figure 66 show the pure query execution performance without TPWSS.

For simple queries ANAPSID is better than FedX in 8/14 queries in contrast to the results on SPARQL1.0 queries. A deeper look into the results shows the reason for ANAPSID's poor performance on SPARQL 1.0 simple queries is due to the time consumed by the source selection. On average, FedX (100% cached) spent only 6ms for the source selection. On the other hand, ANAPSID spent 165ms on average. For 4/14 queries, ANAPSID's source selection time was greater than the rest of the query execution time (excluding source selection). This shows that efficient TPWSS and the corresponding source selection time is of significant importance while dealing with simple queries. In the simple query category, FedX overestimates more than half (142 FedX vs. 67 optimal ref. Table 41) of the sources on average. Thus, by using a perfect TPWSS (i.e., in SPARQL 1.1 version), FedX's performance is improved by 54%. This further shows that the *total triple pattern-wise sources selected* is one of key performance metric missing in the state-of-the-art SPARQL query federation benchmarks. Even though ANAPSID does not overestimate the relevant sources to a large extent (80 ANAPSID vs. 67 optimal), yet its performance is improved by 94% on SPARQL 1.1 versions of the simple queries. The reason is the poor query decomposition plan[7] generated for the SPARQL 1.0 version of queries S6 (308514 ms vs. 1620 ms) and S7 (298954 ms vs. 2157 ms).

For complex queries the ranking is reversed and FedX is better than ANAPSID in 6/8 comparable queries. This result is as expected because FedX overestimated more sources than ANAPSID (199 FedX vs. 111 ANAPSID ref. Table 41). Thus, an optimal TPWSS (as in SPARQL 1.1 version of LargeRDFBench) provides more benefits to FedX. Due to the optimal source selection, FedX's performance is improved by 28.5% and ANAPSID's performance is only improved by 0.8%.

In general, the results above clearly suggests that FedX's performance can be improved significantly by using smart source selection such as join-aware triple pattern-wise source selections as implemented by HiBISCuS and ANAPSID. Furthermore, the metrics *total*

7 Decomposed plans given at: http://goo.gl/AUa0uS

triple pattern-wise sources selected and the corresponding *source selection time*, which were previously ignored, have a significant impact on overall query performance and allow providing tool developers with more fine-grained insights pertaining to their frameworks.

10

FEASIBLE: A FEATURED-BASED SPARQL BENCHMARKS GENERATION FRAMEWORK

This chapter is based on [78] and introduces FEASIBLE, a feature-based SPARQL benchmark generation framework. Triple stores are the data backbone of many Linked Data applications [63]. The performance of triple stores is hence of central importance for Linked-Data-based software ranging from real-time applications [48; 80] to on-the-fly data integration frameworks [1; 77; 94]. Several benchmarks (e.g., [3; 17; 32; 63; 91; 93]) for assessing the performance of the triple stores have been proposed. However, many of them (e.g., [3; 17; 32; 93]) either rely on synthetic data or synthetic queries. The main advantage of synthetic benchmarks is that they commonly rely on data generators that can produce benchmarks of different data sizes and thus allow to test the scalability of triple stores. On the other hand, they often fail to reflect reality. In particular, previous works [25] point out that artificial benchmarks are typically highly structured while real Linked Data sources are most commonly weakly structure. Moreover, synthetic queries most commonly fail to reflect the characteristics of the real queries sent to applications [8; 68]. Thus, synthetic benchmark results are rarely sufficient to detect the most suitable triple store for a particular real application. The DBpedia SPARQL Benchmark (DBPSB) [63] addresses a portion of these drawbacks partly by evaluating the performance of triple stores based on real DBpedia query logs. The main drawback of this benchmark is still that it does not consider important data-driven and structural query features (e.g., number of join vertices, triple patterns selectivities or query execution times etc.) which greatly affect the performance of triple stores [3; 30] during the query selection process. Furthermore, it only considers `SELECT` queries. The other three basic SPARQL query forms, i.e., `ASK`, `CONSTRUCT`, and `DESCRIBE` are not included.

In this chapter we present FEASIBLE, a benchmark generation framework able to generate benchmarks from a set of queries (in particular from query logs). Our approach aims to generate customized benchmarks for given use cases or needs of an application. To this end, FEASIBLE assumes that it is given a set of queries well as the number of queries (e.g., 25) to be included into the benchmark as input. Then, our approach computes a sample of the selected subset that reflects the distribution of the queries in the input set of queries. The resulting queries can then be fed to a benchmark execution framework to benchmark triple stores. The contributions of this work are as follows:

1. We present the first structure and data-driven feature-based benchmark generation approach from real queries. By comparing FEASIBLE with DBPSB, we show that considering data-driven and structural query features leads to benchmarks of better approximation of the input set of queries.

2. We present a novel sampling approach for queries based based on exemplars [66] and medoids.

3. Beside SPARQL `SELECT`, we conduct the first evaluation of 4 triple stores w.r.t. to their performance on `ASK`, `DESCRIBE` and `CONSTRUCT` queries separately.

4. We show that the performance of triple stores varies greatly across the four basic forms of SPARQL query. Moreover, the features used by FEASIBLE allow for a more fine-grained analysis of the results of benchmarks.

The rest of this chapter is structured as follows: We begin by providing an overview of the key SPARQL query features that need to be considered while designing SPARQL benchmarks. Then, we compare existing benchmarks against these key query features systematically (Section 10.2) and point out the weaknesses of current benchmarks that are addressed by FEASIBLE. Our benchmark generation process is presented in Section 10.3. A detailed comparison with DBPSB and an evaluation of the state-of-the-art triple stores follows next. The results are then discussed and we finally conclude. FEASIBLE is open-source and available online at `https://code.google.com/p/feasible/`. A demo can be found at `http://titan.informatik.uni-leipzig.de:8080/feasible/`.

10.1 KEY SPARQL FEATURES

According to previous works [3; 30], a SPARQL query benchmark should vary the queries it contains w.r.t. the following *query characteristics*: number of triple patterns, number of join vertices, mean join vertex degree, query result set sizes, mean triple pattern selectivities, join vertex types ('star', 'path', 'hybrid', 'sink'), and SPARQL clauses used (e.g., `LIMIT`, `OPTIONAL`, `ORDER BY`, `DISTINCT`, `UNION`, `FILTER`, `REGEX`). In addition, a SPARQL benchmark should contain (or provide options to select) all four SPARQL query forms (i.e., `SELECT`, `DESCRIBE`, `ASK`, and `CONSTRUCT`). Furthermore, the benchmark should contain queries of varying runtimes, ranging from small to reasonably large query execution times. In the next section, we compare state-of-the-art SPARQL benchmarks based on these query features.

10.2 A COMPARISON OF EXISTING TRIPLE STORES BENCHMARKS AND QUERY LOGS

Different benchmarks have been proposed to compare triple stores for their query execution capabilities and performance. Table 44 provides a detailed summary of the characteristics of the most commonly used benchmarks as well as of two real query logs. Note of the results of this table is already presented in Chapter 3. We are reusing the text to better motivate the need of comprehensive SPARQL benchmark for triple stores evaluation.

LUBM was designed to test the triple stores and reasoners for their reasoning capabilities. It is based on a customizable and deterministic generator for synthetic data. The queries included in this benchmark commonly lead to query results sizes ranges from 2 to 3200, query triple patterns ranges from 1 to 6, and all the queries consist of a single BGP. LUBM includes a fixed number of `SELECT` queries (i.e., 15) where none of the clauses shown in Table 44 is used.

The Berlin SPARQL Benchmark (BSBM) [17] uses a total of 125 query templates to generate any number of SPARQL queries for benchmarking. Multiple use cases such as explore, update, and business intelligence are included in this benchmark. Furthermore, it also includes many of the important SPARQL clauses of Table 44. However, the queries included in this benchmark are rather simple with an average query runtime of 9.1 ms and largest query result set size equal to 31.

SP^{2}Bench mirrors vital characteristics (such as power law distributions or Gaussian curves) of the data in the DBLP bibliographic database. The queries given in benchmark are mostly complex. For example, the mean (across all queries) query result size is above one million and the query runtimes are very large (see Table 44).

The Waterloo SPARQL Diversity Test Suite (WatDiv) [3] addresses the limitations of previous benchmarks by providing a synthetic data and query generator to generate large number of queries from a total of 125 queries templates. The queries cover both simple and complex categories with varying number of features such as result set sizes, total number of query triple patterns, join vertices and mean join vertices degree. However, this benchmark is restricted to conjunctive `SELECT` queries (single BGPs). This means that non-conjunctive SPARQL queries (e.g., queries which make use of the `UNION` and `OPTIONAL` features) are not considered. Furthermore, WatDiv does not consider other important SPARQL clauses, e.g., `FILTER` and `REGEX`. However, our analysis of the query logs of DBpedia3.5.1 and SWDF given in table 44 shows that 20.1% resp. 7.97% of the DBpedia queries make use of `OPTIONAL` resp. `UNION` clauses. Similarly, 29.5% resp. 29.3% of the SWDF queries contain `OPTIONAL` resp. `UNION` clauses.

While the distribution of query features in the benchmarks presented so far is mostly static, the use of different SPARQL clauses

Table 44: Comparison of SPARQL benchmarks and query logs (**F-DBP** = FEASIBLE Benchmarks from DBpedia query log, **DBP** = DBpedia query log, **F-SWDF** = FEASIBLE Benchmark from Semantic Web Dog Food query log, **SWDF** = Semantic Web Dog Food query log, **TPs** = Triple Patterns, **JV** = Join Vertices, **MJVD** = Mean Join Vertices Degree,**S.D.** = Standard Deviation). Runtime(ms)

		LUBM	BSBM	SP2Bench	WatDiv	DBPSB	F-DBP	DBP	F-SWDF	SWDF
	#Queries	15	125	12	125	125	125	130466	125	64030
Forms (%)	SELECT	100	80	91.67	100	100	95.2	97.9	92.8	58.7
	ASK	0	0	8.33	0	0	0	1.93	2.4	0.09
	CONSTRUCT	0	4	0	0	0	4	0.09	3.2	0.04
	DESCRIBE	0	16	0	0	0	0.8	0.02	1.6	41.1
Clauses (%)	UNION	0	8	16.67	0	36	40.8	7.97	32.8	29.3
	DISTINCT	0	24	41.6	0	100	52.8	4.1	50.4	34.18
	ORDER BY	0	36	16.6	0	0	28.8	0.3	25.6	10.67
	REGEX	0	0	0	0	4	14.4	0.2	16	0.03
	LIMIT	0	36	8.33	0	0	38.4	0.4	45.6	1.79
	OFFSET	0	4	8.33	0	0	18.4	0.0	20.8	0.14
	OPTIONAL	0	52	25	0	32	30.4	20.1	32	29.5
	FILTER	0	52	58.3	0	48	58.4	93.3	29.6	0.72
	GROUP BY	0	0	0	0	0	0.8	7.6E-6	19.2	1.34
Results	Min	3	0	1	0	197	1	1	1	1
	Max	1.3E+4	31	4.3E+7	4.1E+9	4.6E+6	1.4E+6	1.4E+6	3.0E+5	3.0E+5
	Mean	4.9E+3	8.3	4.5E+6	3.4E+7	3.2E+5	5.2E+4	404	9091	39.5
	S.D.	1.1E+4	9.03	1.3E+7	3.7E+8	9.5E+5	1.9E+5	1.2E+4	4.7E+4	2208
BGPs	Min	1	1	1	1	1	1	0	0	0
	Max	1	5	3	1	9	14	14	14	14
	Mean	1	2.8	1.5	1	2.69	3.17	1.67	2.68	2.28
	S.D.	0	1.70	0.67	0	2.43	3.55	1.66	2.81	2.9
TPs	Min	1	1	1	1	1	1	0	0	0
	Max	6	15	13	12	12	18	18	14	14
	Mean	3	9.32	5.9	5.3	4.5	4.8	1.7	3.2	2.5
	S.D.	1.81	5.17	3.82	2.60	2.79	4.39	1.68	2.76	3.21
JV	Min	0	0	0	0	0	0	0	0	0
	Max	4	6	10	5	3	11	11	3	3
	Mean	1.6	2.88	4.25	1.77	1.21	1.29	0.02	0.52	0.18
	S.D.	1.40	1.80	3.79	0.99	1.12	2.39	0.23	0.65	0.45
MJVD	Min	0	0	0	0	0	0	0	0	0
	Max	5	4.5	9	7	5	11	11	4	5
	Mean	2.02	3.05	2.41	3.62	1.82	1.44	0.04	0.96	0.37
	S.D.	1.29	1.63	2.26	1.40	1.43	2.13	0.33	1.09	0.87
MTPS	Min	3.2E-4	9.4E-8	6.5E-5	0	1.1E-5	2.8E-9	1.2E-5	1.0E-5	1.0E-5
	Max	0.432	0.045	0.53	0.011	1	1	1	1	1
	Mean	0.01	0.01	0.22	0.004	0.119	0.140	0.005	0.291	0.0238
	S.D.	0.074	0.01	0.20	0.002	0.22	0.31	0.03	0.32	0.07
Runtime	Min	2	5	7	3	11	2	1	4	3
	Max	3200	99	7.1E+5	8.8E+8	5.4E+4	3.2E+4	5.6E+4	4.1E+4	4.1E+4
	Mean	437	9.1	2.8E+5	4.4E+8	1.0E+4	2242	30.4	1308	16.1
	S.D.	320	14.5	5.2E+5	2.7E+7	1.7E+4	6961	702.5	5335	249.6

and triple pattern join types varies greatly from data set to data set, thus making it very difficult for any synthetic query generator to reflect real queries. For example, the DBpedia and SWDF query log differ significantly in their use of `DESCRIBE` (41.1% for SWDF vs 0.02% for DBpedia), `FILTER` (0.72% for SWDF vs 93.3% for DBpedia) and `UNION` (29.3% for SWDF vs 7.97% for DBpedia) clauses. Similar variations have been reported in [8] as well. To address this issue, the DBpedia SPARQL Benchmark (DBPSB) [63] (which generates benchmark queries from query logs) was proposed. However, this benchmark does not consider key query features (i.e., number of join vertices, mean join vertices degree, mean triple pattern selectivities, the query result size and overall query runtimes) while selecting query templates. Note that previous works [3; 30] pointed that these query features greatly affect the triple stores performance and thus should be considered while designing SPARQL benchmarks.

In this work we present FEASIBLE, a benchmark generation framework which is able to generate a customizable benchmark from any set of queries, esp. from query logs. FEASIBLE addresses the drawbacks on previous benchmark generation approaches by taking all of the important SPARQL query features of Table 44 into consideration when generating benchmarks. In the following, we present our approach in detail.

10.3 FEASIBLE BENCHMARK GENERATION

The benchmark generation behind our approach consists of 3 main steps. The first step is the cleaning step. Thereafter, the features of the queries are normalized. In a final step, we then select a sample of the input queries that reflects the cleaned input queries and return this sample. The sample can be used as seed in template-based benchmark generation approaches such as DBSBM and BSBM.

10.3.1 *Data Set Cleaning*

The aim of the data cleaning step is to remove erroneous and zero-result queries from the set of queries used to generate benchmarks. This step is not of theoretical necessity but leads to practically reliable benchmarks. To clean the input data set (here query logs), we begin by excluding all syntactically incorrect queries. The syntactically correct queries which lead to runtime errors[1] as well as queries which return zero results are removed from the set of relevant queries for benchmarking. We attach all 9 SPARQL clauses (e.g., `UNION`, `DISTINCT`) and 7 query features (i.e., runtime, join vertices, etc.) given in Table 44 to each of the queries. For the sake of simplicity we call these 16 (i.e.,

1 The runtime errors were measured using Virtuoso 7.2.

9+7) properties *query features* in the following . All unique queries are then stored in a file[2] and given as input to the next step.

10.3.2 *Normalization of Features Vectors*

The query selection process of FEASIBLE demands computing distances between queries. To ensure that dimensions with high values (e.g., the result set size) do not bias the selection, we normalize the query representations to ensure that all queries are located in a unit hypercube. To this end, each of the queries gathered from the previous step is mapped to a vector of length 16 which stores the corresponding *query features* as follows: For the SPARQL clauses, which are binary (e.g., `UNION` is either used or not used), we store a value 1 if that clause in used in the query. Otherwise we store a 0. All non-binary features vectors are normalized by dividing their value with the overall maximal value in the data set. Therewith, we ensure that all entries of the query representations are values between 0 to 1.

10.3.3 *Query Selection*

The query selection process is based on the idea of exemplars used in [66] and is shown in Algorithm 9. We assume that we are given (1) a number $e \in \mathbb{N}$ of queries to select as benchmark queries as well as (2) a set of queries L with $|L| = n >> e$, where L is the set of all cleaned and normalized queries. The intuition behind our selection approach is to compute an e-sized partition $\mathcal{L} = \{L_1, \ldots, L_e\}$ of L that is such that (1) the average distance between the points in two different elements of the partition is high and (2) the average distance of points within a partition is small. We can then select the point closest to the average of each L_i (i.e., the medoid of L_i) to be a prototypical example of a query from L and include it into the benchmark generated by FEASIBLE. We implement this intuition formally by (1) selecting e exemplars (i.e., points that represent a portion of the space) that are as far as possible from each other, (2) partitioning L by mapping every point of L to one of these exemplars to compute a partition of the space at hand and (3) selecting the medoid of each of the partitions of space as a query in the benchmark. In the following, we present each of these steps formally. For the sake of clarity, we use the following running example: $L = \{q_1 = [0.2, 0.2], q_2 = [0.5, 0.3], q_3 = [0.8, 0.5], q_4 = [0.9, 0.1], q_5 = [0.5, 0.5]\}$ and assume that we need a benchmark with $e = 2$ queries. Note for the sake of simplicity, we used normed features vectors of length 2 instead of 16.

2 A sample file can be found at `http://goo.gl/YUSU9A`

Algorithm 9 Query Selection Approach

Require: Set of queries L; Size of the benchmark e

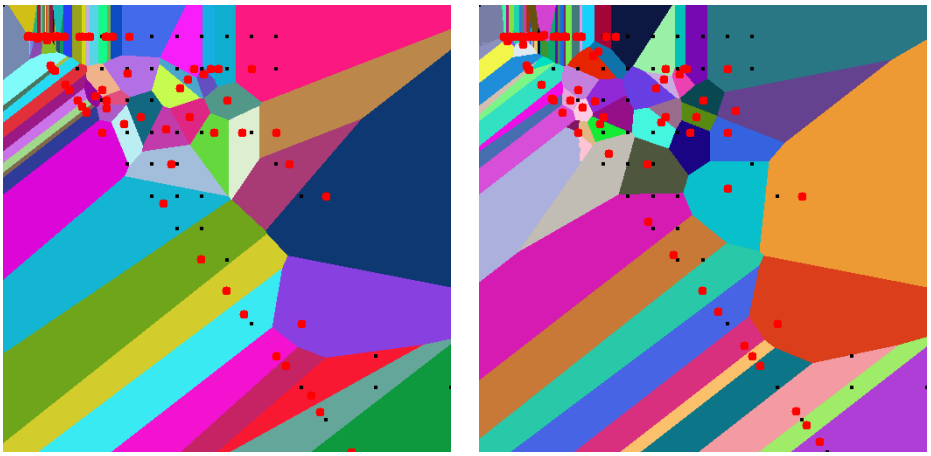
1: $\tilde{L} = \frac{1}{|L|} \sum_{q \in L} q$
2: $X_1 = \{\arg\min_{x \in L} d(\tilde{L}, x)\}$
3: $\mathcal{X} = \{X_1\}$
4: **for** $i = 2; i \leqslant e; i++$ **do**
5: $\quad X_i = \{\arg\max_{y \in L \setminus \mathcal{X}} d(y, \mathcal{X})\}$
6: $\quad \mathcal{X} = \mathcal{X} \cup \{X_i\}$
7: **end for**
8: $\mathcal{L} = \emptyset$
9: **for** $i = 1; i \leqslant e; i++$ **do**
10: $\quad L_i = \{X_i\}$ $\mathcal{L} = \mathcal{L} \cup \{L_i\}$
11: **end for**
12: **for** $i = 1; i \leqslant e; i++$ **do**
13: $\quad L_i = \{q \in L \setminus \mathcal{X} : X_i = \arg\min_{X \in \mathcal{X}} d(X, q)\}$
14: **end for**
15: $B = \emptyset$
16: **for** $i = 1; i \leqslant e; i++$ **do**
17: $\quad \tilde{L_i} = \frac{1}{|L_i|} \sum_{q \in L_i} q$
18: $\quad b_i = \arg\min_{q \in L_i} d(\tilde{L_i}, q)$
19: $\quad B = B \cup \{b_i\}$
20: **end for**
21: **return** B

10.3.3.1 *Selection of Exemplars*

We implement an iterative approach to the selection of exemplars (see lines 1-7 of Algorithm 9). We begin by finding the average $\tilde{L} = \frac{1}{n} \sum_{q \in L} q$ of all representations of queries $q \in L$. In our example, this point has the coordinates $[0.58, 0.32]$. The first exemplar X_1 is the point of L that is closest to the average and is given by $X_1 = \arg\min_{x \in L} d(\tilde{L}, x)$, where d stands for the Euclidean distance. In our example, this is the query q_2 with a distance of 0.08. We follow an iterative procedure to extending the set $\mathcal{X}$ of all exemplars: We first find $\eta = \arg\max_{y \in L \setminus \mathcal{X}} \left(\sum_{x \in \mathcal{X}} d(x, y) \right)$. η is the point that is furthest away from all exemplars. In our example, that is the query q_4 with a distance of 0.45 from q_2. We then add η to $\mathcal{X}$ and repeat the procedure for finding η until $|\mathcal{X}| = e$. Given that $e = 2$ in our example, we get the set $\mathcal{X} = \{q_2, q_4\}$ as set of exemplars.

10.3.3.2 *Selection of Benchmark Queries*

Let $\mathcal{X} = \{X_1, \ldots, X_e\}$ the set of all exemplars. The selection of benchmark queries begins with partitioning the space according to $\mathcal{X}$. The partition L_i is defined as $L_i = \{q \in L : \forall i \neq j : d(q, X_i) \leqslant d(q, X_j)\}$ ((see lines 8-15 of Algorithm 9). It is simply the set of queries that are closer to X_i than to any other exemplar. In case of a tie, i.e., $d(q, X_i) = d(q, X_j)$ with $i \neq j$, we assign q to $\min(i, j)$. In our example, we get the following partition: $\mathcal{X} = \{\{q_1, q_2, q_3, q_5\}, \{q_4\}\}$. Finally,



(a) DBpedia-125 (b) DBpedia-175

Figure 70: Voronoi diagrams for benchmarks generated by FEASIBLE along the two axes with maximal entropy. Each of the red points is a benchmark query. Several points are superposed as the diagram is a projection of a 16-dimensional space unto 2 dimensions.

we perform the selection of prototypical queries from each partition (see lines 17-22 of Algorithm 9). For each partition L_i we begin by computing the average $\tilde{L_i}$ of all representations of queries in L_i. We then select the query $b_i = \arg\min_{q \in L_i} d(\tilde{L_i}, q)$. The set B of benchmark queries is the set of all queries b_i over all L_i. Note that $|B| = e$. In our example, q_4 being the only query in the second partition means that q_4 is selected as representative for the second partition. The average of the first partition is located at $[0.5, 0.375]$. The query q_2 is the closest to the average, leading to q_2 being selected as representative for the first partition. Our approach thus returns a benchmark with the queries $\{q_2, q_4\}$ as result.

Figures 70a and 70b show Voronoi diagrams of the results of our approach for benchmarks of size 125 and 175 derived from the DBpedia 3.5.1 query log presented in Table 44 along the two dimensions with the highest entropy. Note that some of the queries are superposed in the diagram.

10.4 COMPLEXITY ANALYSIS

In the following, we study the complexity of our benchmark generation approach. We denote the number of features considered during the generation process with d. e is the number of exemplars and $|L|$ the size of the input data set.

Reading and cleaning the file can be carried out in $O(|L|d)$ as each query is read once and the features are extracted one at a time. We now need to *compute the exemplars*. We begin by computing the average A of all queries, which can be carried out using $O(|L|d)$ arith-

metic operations. Finding the query that is nearest to A has the same complexity. The same approach is used to detect the other exemplars, leading to an overall complexity of $O(e|L|d)$ for the computation of exemplars. *Mapping each point* to the nearest exemplar has an a-priori complexity of $O(e|L|d)$ arithmetic operations. Given that the distances between the exemplars and all the points in L are available from the previous step, we can simply look up the distances and thus gather this information in $O(1)$ for each pair of exemplar and point, leading to an overall complexity of $O(e|L|)$. Finally, *the selection of the representative in the cluster* demands averaging the elements of the cluster and selecting the query that is closest to this point. For each cluster of size $|Cl|$, we need $(d|Cl|)$ arithmetic operations to find the average point. The holds for finding the query nearest to the average. Given that the sum of the sizes of all the clusters is $|L|$, we can conclude that the overall complexity of the selection step is $O(d|L|)$. Overall, the worst-case complexity of our algorithm is thus $O(d|L||E|)$.

In the best case, no queries passes the cleaning test, leading to no further processing and to the same complexity as reading the data, which is $O(|L|d)$. The same best-case complexity holds when a benchmark is generated. Here, the filtering step returns exactly e queries, leading to the exemplar generation step being skipped and thus to a complexity of $O(|L|d)$.

10.5 EVALUATION AND RESULTS

Our evaluation comprises two main parts. First, we compare FEASIBLE with DBPSB w.r.t. how well the benchmarks represent the input data. To this end, we use the composite error function defined below. In the second part of our evaluation, we use FEASIBLE benchmarks to compare triple stores for their query execution performance.

10.5.1 *Composite Error Estimation*

The benchmarks we generate aim to find typical queries for a given query log. From the point of view of statistics, this is equivalent to computing a subset of a population that has the same characteristics (here mean and standard deviation) as the original population. Thus, we measure the quality of the sampling approach of a benchmark by how much the mean and standard deviation of the features of its queries deviates from that of the query log. We call μ_i resp. σ_i the mean resp. the standard deviation of a given distribution w.r.t. to the i^{th} feature of the said distribution. Let B be a benchmark extracted

from a set of queries L. We use two measures to compute the similarity of B and L. The error on the means E_μ and deviations E_σ

$$E_\mu = \frac{1}{k}\sum_{j=1}^{k}(\mu_i(L) - \mu_i(B))^2 \text{ and } E_\sigma = \frac{1}{k}\sum_{j=1}^{k}(\sigma_i(L) - \sigma_i(B))^2. \quad (8)$$

We define a composite error estimation E as the harmonic mean of E_μ and E_σ:

$$E = \frac{2E_\mu E_\sigma}{E_\mu + E_\sigma}. \quad (9)$$

10.5.2 *Experimental Setup*

DATA SETS AND QUERY LOGS: We used the DBpedia 3.5.1 (232.5M triples) and SWDF (294.8K triples) data sets for triple stores evaluation. As queries (see Section 10.2), we used 130,466 cleaned queries for DBpedia and 64,029 cleaned queries for SWDF.

BENCHMARKS FOR COMPOSITE ERROR ANALYSIS: In order to compare FEASIBLE with DBPSB, we generated benchmarks of sizes 15, 25, 50, 75, 100, 125, 150, and 175 queries from the DBpedia 3.5.1 query log. Recall this is exactly the same query log used in DBPSB. DBPSB contains a total of 25 query templates derived from 25 real queries. A single query was generated per query template in order to generate a benchmark of 25 queries. Similarly, 2 queries were generated per query template for a benchmark of 50 queries and so on. The 15 queries benchmark of DBPSB was generated from the 25-query benchmark by randomly choosing 15 of the 25 queries. We chose to show results on a 15-query benchmark because LUBM contains 15 queries while SP^2Bench contains 12. We also generated the benchmarks of the same size (15-175) from SWDF to compare FEASIBLE's composite errors as well as the performance of triple stores across different data sets.

TRIPLE STORES: We used four triple stores in our evaluation: (1) *Virtuoso Open-Source Edition version 7.2* with NumberOfBuffers = 680000, MaxDirtyBuffers = 500000; (2) *Sesame Version 2.7.8* with Tomcat 7 as HTTP interface and native storage layout. We set the spoc, posc, opsc indices to those specified in the native storage configuration. The Java heap size was set to 6GB; (3) *Jena-TDB (Fuseki) Version 2.0* with a Java heap size set to 6GB and (4) *OWLIM-SE Version 6.1* with Tomcat 7.0 as HTTP interface. We set the entity index size to 45,000,000 and enabled the predicate list. The rule set was empty and the Java heap size was set to 6GB. Ergo, we configured all triple stores to use 6GB of memory and used default values otherwise.

BENCHMARKS: Most of the previous evaluations were carried out on SELECT queries only (see Table 44). Here, beside evaluating the performance of triples stores on SELECT evaluation, we also wanted to compare triple stores on the other three forms of SPARQL queries. To this end, we generated DBpedia-ASK-100 (100-ASK-query benchmark derived from DBpedia) and SWDF-ASK-50 (50-ASK-query benchmark derived from SWDF)[3] and compared the selected triple stores for their ASK query processing performances. Similarly, we generated DBpedia-CONSTRUCT-100 and SWDF-CONSTRUCT-23, DBpedia-DESCRIBE-25 and SWDF-DESCRIBE-100, and DBpedia-SELECT-100 and SWDF-SELECT-100 benchmarks to test the selected systems for CONSTRUCT, DESCRIBE, and SELECT queries, respectively. Furthermore, we generated DBpedia-Mix-175 (DBpedia benchmark of 175 mix queries of all the four query forms) and SWDF-Mix-175 to test the selected triple stores for their general query processing performance.

BENCHMARK EXECUTION: The evaluation was carried out one triple store at a time on one machine. First, all data sets were loaded into the selected triple store. Once the triple store had completed the data loading, the 2-phase benchmark execution phase began: (1) *Warm-up Phase*: To measure the performance of the triple store under normal operational conditions, a warm-up phase was used where random queries from the query log were posed to triple stores for 10 minutes; (2) *Hot-run Phase:* During this phase, the benchmark query mixes were sent to the tested store. We kept track of the average execution time of each query as well as of the number of query mixes per hour (QMpH). This phase lasted for two hours for each triple store. Note that the benchmark and the triple store were run on the same machine to avoid network latency. We set the query timeout to 180 seconds. The query was aborted after that and maximum time of 180 seconds was used as the query runtime for all queries which timed out. All the data (data dumps, benchmarks, query logs, FEASIBLE code) to repeat our experiments along with complete evaluation results are available at the project website.

10.5.3 *Experimental Results*

10.5.3.1 *Composite Error*

Table 45 shows a comparison of the composite errors of DBPSB and FEASIBLE for different benchmarks. Note that DBPSB queries templates are only available for the DBpedia query log. Thus, we were not able to calculate DBPSB's composite errors for SWDF. As an overall composite error evaluation, FEASIBLE's composite error is

3 We chose to select only 50 queries because the SWDF log we used does not contain enough ASK queries to generate a 100 query benchmark.

54.9% smaller than DBPSB. The reason for DBPSB's error being higher that FEASIBLE's lies in the fact that it only considers the number of query triple patterns and the SPARQL clauses `UNION`, `OPTIONAL`, `FILTER`, `LANG`, `REGEX`, `STR`, and `DISTINCT` as features. Important query features (such as query result sizes, execution times, triple patterns and join selectivities, and number of join vertices) were not considered when generating the 25 queries templates[4]. Furthermore, DBPSB only includes `SELECT` queries. The other three SPARQL query forms, i.e., `CONSTRUCT`, `ASK`, and `DESCRIBE` are not considered. In contrast, our approach considers all of the query forms, SPARQL clauses, and query features reported in Table 44.[5] It is important to mention that FEASIBLE's overall composite error across both data sets is only 0.038.

4 Queries templates available at: http://goo.gl/1oZCZY
5 See FEASIBLE online demo for the customization of these features

Table 45: Comparison of the Mean E_μ , Standard Deviation E_σ and Composite E errors for different benchmark sizes of DBpedia and Semantic Web Dog Food query logs. FEASIBLE outperforms DBPSB across all dimensions.

Benchmark	FEASIBLE			DBPSB			Benchmark	FEASIBLE		
	E_μ	E_σ	E	E_μ	E_σ	E		E_μ	E_σ	E
DBpedia-15	0.045	0.054	0.049	0.139	0.192	0.161	SWDF-15	0.019	0.043	0.026
DBpedia-25	0.041	0.054	0.046	0.113	0.139	0.125	SWDF-25	0.034	0.051	0.041
DBpedia-50	0.045	0.056	0.050	0.118	0.132	0.125	SWDF-50	0.036	0.052	0.043
DDBpedia-75	0.053	0.061	0.057	0.096	0.095	0.096	SWDF-75	0.035	0.051	0.042
DDBpedia-100	0.054	0.064	0.059	0.130	0.132	0.131	SWDF-100	0.036	0.050	0.042
DDBpedia-125	0.054	0.064	0.058	0.088	0.082	0.085	SWDF-125	0.034	0.048	0.040
DBpedia-150	0.055	0.064	0.059	0.107	0.124	0.115	SWDF-150	0.033	0.046	0.038
DBpedia-175	0.055	0.065	0.059	0.127	0.144	0.135	SWDF-175	0.033	0.045	0.038
Average	**0.050**	**0.060**	**0.055**	0.115	0.130	0.121	Average	0.032	0.048	0.039

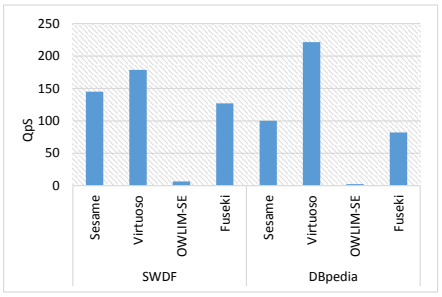
10.5.3.2 *Triple Store Performance*

Figure 71 shows a comparison of the selected triple stores in terms of *queries per second* (QpS) and *query mixes per hour* (QMpH) for different benchmarks generated by FEASIBLE. Table 46 shows the overall rank-wise query distributions of the triple stores. Our ranking is partly different from the DBPSB ranking. Overall, (for mix DBpedia and SWDF benchmarks of 175 queries each, Figure 71e to Figure 71g), Virtuoso ranks first followed by Fuseki, OWLIM-SE, and Sesame. Virtuoso is 59% faster than Fuseki. Fuseki is 1.7% faster than OWLIM-SE, which in turn 16% faster than Sesame. [6]

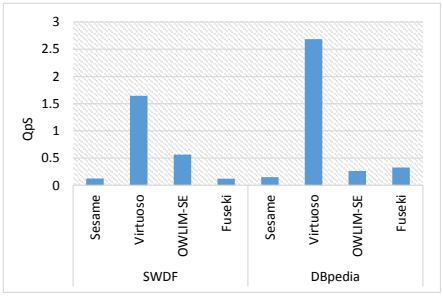
A more fine-grained look at the evaluation reveals surprising findings: On `ASK` queries, Virtuoso is clearly faster than the other frameworks (45% faster than Sesame, which is 16% faster than Fuseki, which is in turn 96% faster than OWLIM-SE, see Figure 71a). The ranking changes for `CONSTRUCT` queries: While Virtuoso is still first (87% faster than OWLIM-SE), OWLIM-SE is now faster that 14% faster than Fuseki, which in turn is 42% faster than Sesame (Figure 71b). The most drastic change occurs on the `DESCRIBE` benchmark, where Fuseki ranks first (66% faster than Virtuoso, which is 86% faster than OWLIM-SE, which in turns 47% faster than Sesame, see Figure 71c). Yet another ranking emerges from the `SELECT` benchmarks, where Virtuoso is overall 55% faster than OWLIM-SE, which is 41% faster than Fuseki, which in turns 11% faster than Sesame (Figure 71d). These results show that the performance of triple stores varies greatly across the four basic SPARQL forms and none of the system is the sole winner across all query forms. Moreover, the ranking also varies across the different datasets (see, e.g., `ASK` benchmark for DBpedia and SWDF). Thus, our results suggest that (1) a benchmark should comprise a mix of SPARQL `ASK`, `CONSTRUCT`, `DESCRIBE`, and `SELECT` queries that reflects the real intended usage of the triple stores to generate accurate results and (2) there is no universal winner amongst triple stores, which points again towards the need to create customized benchmarks for applications when choosing their backend. FEASIBLE addresses both of these requirements by allowing users to generate dedicated benchmarks from their query logs.

Some interesting observations were revealed by the rank-wise queries distributions of triple stores shown in Table 46: First, none of the system is sole winner or loser for a particular rank. Overall, Virtuoso's performance mostly lies in the higher ranks, i.e., rank 1 and 2 (68.29%). This triple stores performs especially well at `CONSTRUCT` queries. Fuseki's performance is mostly in the middle ranks, i.e., rank 2 and 3 (65.14%). In general, it is faster for `DESCRIBE` queries and is on a slower side for `CONSTRUCT` and queries containing `FILTER` and `ORDER BY` clauses. While OWLIM-SE's performance is usually on the

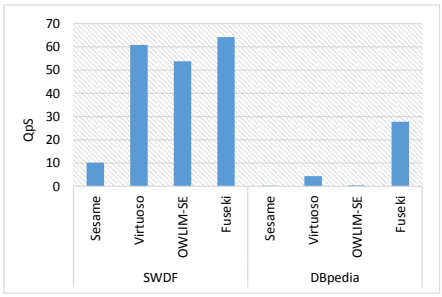
6 Note the percentage improvements are calculated from the QMpH values as A is (1-QMpH(A)/QMpH(B)*100) percent faster than B.



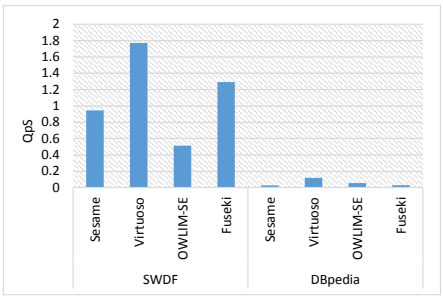
(a) QpS (ASK-Only)



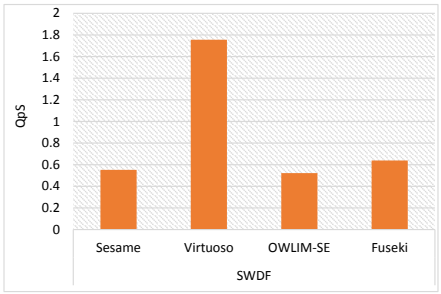
(b) QpS (CONSTRUCT-Only)



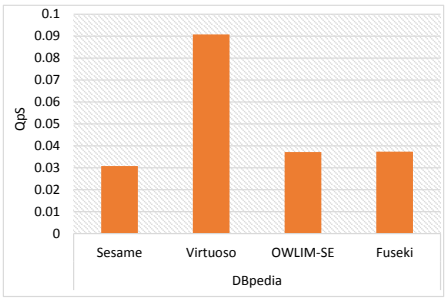
(c) QpS (DESCRIBE-Only)



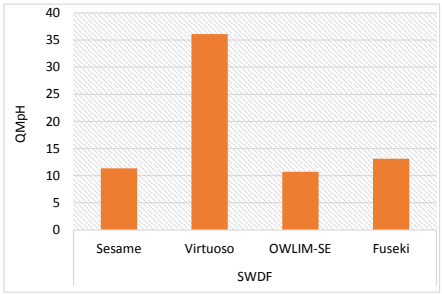
(d) QpS (SELECT-Only)



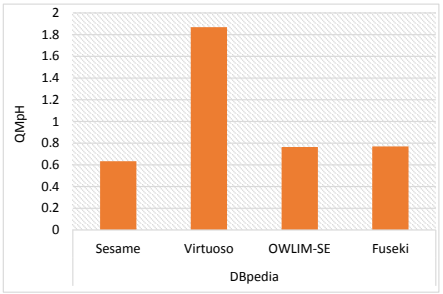
(e) QpS (SWDF-Mix)



(f) QpS (DBpedia-Mix)



(g) QMpH (SWDF-Mix)



(h) QMpH (DBpedia-Mix)

Figure 71: Comparison of the triple stores in terms of Queries per Second (QpS) and Query Mix per Hour (QMpH), where a Query Mix comprise of 175 distinct queries.

slower side, i.e., rank 3 and 4 (60.86 %), it performs well on complex queries with large result set sizes and complex SPARQL clauses. Finally, Sesame is either fast or slow. For example, for 31.71% of the queries, it achieve the rank 1 (second best after Virtuoso) and but achieves rank 4 on 23.14% of the queries (Second worse after OWLIM-SE). In general Sesame is very efficient on simple queries with small result set sizes, a small number of triple triple patterns, and a few SPARQL clauses. However, it performs poorly as soon as the queries grow in complexity. These results shows yet another aspect of the importance of taking structural and data-driven features into consideration while generating benchmarks as they allow deeper insights into the type of queries on which systems perform well or poorly.

Table 46: Overall rank-wise ranking of triple stores. All values are in percentages.

Triple Store	SWDF				DBpedia				Overall			
	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th
Virtuoso	38.29	24.57	21.71	15.43	54.86	18.86	15.43	10.86	46.57	21.71	18.57	13.14
Fuseki	17.14	39.43	32.00	11.43	24.00	34.86	24.00	17.14	20.57	37.14	28.00	14.29
OWLIM-SE	10.29	30.29	21.14	38.29	13.14	24.57	25.14	37.14	11.71	27.43	23.14	37.71
Sesame	37.71	12.00	29.14	21.14	25.71	16.57	32.57	25.14	31.71	14.29	30.86	23.14

Finally, we also looked into the number of query timeouts during the complete evaluation. Most of the systems timeout for `SELECT` queries. Overall, Sesame has the highest number of timeouts (43) followed by Fuseki (32), OWLIM-SE (22), and Virtuoso (14). For Virtuoso, the timeout queries have at least one triple pattern with an unbound subject, an unbound predicate and an unbound object. The corresponding result sets were so large that they could not be computer in 3 minutes. The other three systems mostly timeout for the same queries. OWLIM-SE generally performs better for complex queries with large result set sizes. Fuseki has problems with queries containing `FILTER` (12/32) and `ORDER BY` clauses (11/32 queries). Sesame performs slightly poorer for complex queries containing many triple patterns and joins as well as complex SPARQL clauses. Note that Sesame also times out for 8 `CONSTRUCT` queries. All the timeout queries for each triple store are provided at project website.

11 CONCLUSION

In this chapter we summarize our research work, highlight our core contributions and give general conclusions and future directions.

Overall, our source selection results suggest that the join-aware TPW source selection (as implemented by HiBISCuS, TBSS, SAFE, and TopFed) is the superior paradigm when performing source selection for SPARQL endpoint federation. Our benchmark evaluation clearly indicates that while current federation engines can deal with simple and complex queries, they are currently not up to the challenge of dealing with real queries that involve processing large intermediate result sets or lead to large result sets. In addition, one-fits-all solutions do not work when benchmarking towards a given use case and benchmarks must look at all types of SPARQL constructs. In the following, we provide more details pertaining to these insights.

11.1 HIBISCUS

We presented HiBISCus, a novel labelled-hypergraph-based approach for efficient source selection for SPARQL endpoint federation. HiBISCuS makes use of URI authorities to only select those sources that contribute the final result set of any given query. We evaluated our approach against DARQ, SPLENDID, FedX and ANAPSID. The evaluation shows that the query runtime of these systems is improved significantly.

In future, the impact of the threshold θ on our approach will be investigated. The effect of our source pruning algorithm on SPARQL 1.1 queries with SPARQL *service clause* will be studied, where the TPW sources are already specified by the user. Furthermore, our approach can be evaluated on big data as the query execution time for majority of the FedBench queries is less than 1s, which makes it difficult to select the best SPARQL federation engine and have a deeper look into the behaviour of these engines in different data environments.

11.2 TBSS/QUETSAL

We have seen that HiBISCuS can significantly remove irrelevant sources. However, it fails to prune those sources which share the same URI authority. We have addressed some of limitations HiBISCuS in TBSS by using common name spaces instead of URIs authorities. In addition, we combined HiBISCuS, TBSS, and DAW with global SPARQL 1.1 query rewriting into QUETSAL, a complete SPARQL query federa-

tion engine. We evaluated QUETSAL against state-of-the-art federation systems. The evaluation shows that QUETSAL outperforms the state-of-the-art by reducing the number of sources selected and generating a considerable number of remote joins, a key to federated query processing optimization.

In future,QUETSAL will be improved further by using bind joins (as used in state-of-the-art engines) to solve the problem of retrieving many intermediate results for queries where many SPARQL `UNION` clauses are introduced by the SPARQL 1.1 query rewrite.

11.3 DAW

In this thesis we presented DAW, an approach for duplicate-aware federated query over the Web of Data. DAW combines min-wise independent permutations with selectivity values to estimate the number of duplicate-free results. This estimation is used to first rank triple pattern-wise sources, based on their contribution, and to skip sources that contribute with little or no new results. DAW will be directly combined with existing index-assisted federated query processing systems, in order to improve the query execution. We evaluated our approach against DARQ, SPLENDID and FedX – three well known federated systems. The evaluation shows that by using the DAW extension the query execution times were improved in most of the cases, while recall was marginally affected. Moreover, DAW is suitable for maximising the recall for a fixed number of queried sources.

In the future, DAW index will be extended to further reduce the query execution time, for instance, by pre-computing some of the overlap statistics, based on query logs. The effect of different MIPs sizes and threshold values to find the optimal trade-off between execution time and recall will be explored, as well as different data partition methods.

11.4 SAFE

We have presented SAFE: a query federation engine that enables policy-based access to sensitive statistical datasets represented as RDF Data Cubes. The work is motivated in particular by the needs of three clinical organisations who wish to develop a platform for collaboratively analysing clinical data that spans multiple clinical sites, thus improving the statistical power of conclusions that can be drawn (versus one source alone). Clinical data – even in aggregated form – is of a highly sensitive nature, and thus federated querying methods must take access policies into account. SAFE is developed as an extension on top of the FedX federation engine to support two main features: (i) optimisations tailored for federating querying of RDF Data Cubes; and (ii) source-selection on the level of named graphs that allows for

integration with an existing access-control layer. We evaluated these extensions based on our internal data sets (private data owned by clinical organisations) as well as external data sets (public data available from the LOD cloud) in order to measure the efficiency of SAFE against FedX. Our evaluation results show that, for our use-case(s), SAFE outperforms FedX in terms of fastest source selection and query execution time. SAFE will further be extended to work on top of any datasets, instead of RDF data cubes.

11.5 TOPFED

In this work, we have published a Linked Data version of TCGA data level 3 (to the best of our knowledge the largest Linked Data dump anywhere) and further linked it to the LOD cloud. This big data resource is designed to be used as infrastructure for biomedical and bioinformatics applications that analyse and query both the file annotations but also the internal content of the patient-derived files of this key reference for molecular biology and epidemiology of cancer. The TCGA data dump (and what we expect will be the genomics datasets in the future) is already too large to be effectively handled by a single server. If the relationships between TCGA and other related resources are taken into account, a smart data distribution framework that distributes the data among multiple SPARQL endpoints, such as the one reported here is, an absolute necessity. This framework, TopFed, is specifically designed as a federated query processing engine that handles a collection of physically distributed RDF data sources. The resulting virtually integrated data resource was observed to enable significantly faster querying and retrieval (one third) than current solutions, such as FedX. The TopFed source selection algorithm achieves this result by considering the metadata about the data distribution with the type of the joins among query triples patterns. The substantial improvements in efficient processing achieved, also in the use of network traffic, suggests that the development of systems designed to process an individual patient clinical data to identify the drugs leading to better outcomes in related cohorts in TCGA-like resources (e.g., ICGC [1]) is now at hand.

One of our future aims is to develop an intelligent system, in which a cancer patient's genomic data are used as input to suggest effective drugs for treatment while comparing against results from TCGA patients with the same or similar cancer sub-types. In 2009, we contributed to CNViewer [2], a browser based tool that could be used, via oncologists uploading their own patient's copy number result, to calculate the Euclidean (or other) distance to all other patients with the same tumour type. With TopFed, not only we can calculate these dis-

1 http://icgc.org/
2 https://sites.google.com/site/cnviewerguide/

tances using copy number results, but in future work we expect to use aggregation/correlation of molecular results to match and better understand both the biology driving cancer and the most effective treatment for a patient given a set of genetic alterations.

11.6 LARGERDFBENCH

In this thesis we presented LargeRDFBench, the first billion-triple benchmark for federated SPARQL query engines based on real data and real queries. We presented the three different types of queries contained in the benchmark and compared state-of-the-art systems against these queries. Our evaluation clearly indicates that while current systems can deal with simple and complex queries, they are currently not up to the challenge of dealing with real large result set queries. Alarmingly, the systems return partly incomplete results without making the user aware of this incompleteness. In the future, triple stores such as Virtuoso, Sesame etc. that supports SPARQL 1.1 federated queries will be tested with this benchmark.

LargeRDFBench will further be extended to deal with data partitioning, network traffic, and dynamic data updates etc.

11.7 FEASIBLE

Finally, we presented FEASIBLE, a customizable SPARQL benchmark generation framework. We compared existing SPARQL benchmarks by presenting the detail query statistics of each of the benchmark. Our evaluation showed that existing benchmarks range from very simple (e.g., BSBM) to very complex (e.g., SP^{2}Bench). Some of them (e.g., WatDiv and LUBM) only focus on `SELECT` and conjunctive queries. We then showed that it is very difficult for artificial SPARQL benchmark generation frameworks to reflect the characteristics of real query logs. We compared FEASIBLE with DBPSB and showed that our approach is able to produce high-quality (in terms of small composite error) benchmarks. In addition, our framework allows users to generate customized benchmarks suited for a particular use case, which is of utmost importance when aiming to gather valid insights into the real performance of different triple stores for a given application. This is demonstrated by our triple stores evaluation, which shows that the ranking of triple stores varies greatly across different types of queries as well as across datasets. Our results thus suggest that all of the four query forms should be included in the future SPARQL benchmarks.

For the sake of future work, FEASIBLE will be extended to attach many of the SPARQL 1.1 features to each of the query, thus allow user to generate customized SPARQL 1.1 benchmarks.

As future vision of federated SPARQL queries, running federated queries over large datasets, e.g., the Linked TCGA SPARQL endpoints[3] may result in millions of results with queries runtimes in hours. In such cases, it is possible that a user may not be interested in the complete results. Rather, a subset of results in a reasonable amount of query runtime might be more important for the query executor. Existing SPARQL query federation engines focused on the problem of generating optimized query execution plans. None, to the best of our knowledge, has taken in to account the time efficient Top-K results retrieval which might be important for a particular use case. Similarly, Top-K results retrieval and query personalization based on user profile and location is interesting research direction in federated SPARQL query processing. Optimization based on query tuning has a great potential to improve state-of-the-art work. Collecting provenance information (e.g., how many results are contributed by each data source) and estimating the query runtime before execution might be useful information for the end users. Data distribution-aware federated, non to the best of our knowledge, query engines have the great potential to outperform state-of-the-art engines. Data de-duplication at runtime from federated data sources is interested topic that can be investigated.

3 Linked TCGA SPARQL endpoints: http://tcga.deri.ie/

BIBLIOGRAPHY

[1] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. Anapsid: an adaptive query processing engine for sparql endpoints. In *ISWC*, 2011. (Cited on pages 6, 17, 18, 20, 29, 31, 34, 35, 59, 81, 89, 91, 118, and 175.)

[2] Z. Akar, T. G. Halaç, E. E. Ekinci, and O. Dikenelli. Querying the web of interlinked datasets using void descriptions. In *LDOW at WWW*, 2012. (Cited on pages 29, 30, and 31.)

[3] G. Aluç, O. Hartig, M. T. Ozsu, and K. Daudjee. Diversified stress testing of rdf data management systems. In *ISWC*, 2014. (Cited on pages 3, 14, 23, 153, 155, 175, 176, 177, and 179.)

[4] F. Amorim. Join reordering and bushy plans. In *Simple Talk: A technical journal and community hub from Red Gate*, 2013. (Cited on page 55.)

[5] C. H. Andriy Nikolov, Andreas Schwarte. Fedsearch: efficiently combining structured queries and full-text search in a sparql federation. In *ISWC*. 2013. (Cited on pages 29, 31, and 113.)

[6] A. Antoniades, J. A. Keane, A. Aristodimou, C. Philipou, A. Constantinou, C. Georgousopoulos, F. Tozzi, K. C. Kyriacou, A. Hadjisavvas, M. Loizidou, C. Demetriou, and C. S. Pattichis. The effects of applying cell-suppression and perturbation to aggregated genetic data. In *BIBE*, pages 644–649. IEEE Computer Society, 2012. (Cited on page 115.)

[7] M. Arenas and J. Pérez. *Federation and Navigation in SPARQL 1.1*. Springer, 2012. (Cited on pages 11 and 13.)

[8] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. *CoRR*, 2011. (Cited on pages 3, 23, 24, 153, 156, 175, and 179.)

[9] S. Auer, J. Lehmann, and A.-C. Ngonga Ngomo. Introduction to linked data and its lifecycle on the web. In *Reasoning Web*, pages 1–75, 2011. (Cited on page 97.)

[10] E. Bair, T. Hastie, D. Paul, and R. Tibshirani. Prediction by supervised principal components. *Journal of the American Statistical Association*, 101(473), 2006. (Cited on page 127.)

[11] C. Basca and A. Bernstein. Avalanche: putting the spirit of the web back into semantic web querying. In *SSWS*, pages 64–79, November 2010. (Cited on pages 29, 30, and 31.)

[12] D. Beckett. RDF/XML Syntax Specification (Revised), 10. Februar 2004. W3C Recommendation. (Cited on page 10.)

[13] G. Bell, T. Hey, and A. Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009. (Cited on page 127.)

[14] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001. (Cited on page 9.)

[15] H. Betz, F. Gropengießer, K. Hose, and K.-U. Sattler. Learning from the history of distributed query processing - a heretic view on linked data management. In *COLD*, 2012. (Cited on page 20.)

[16] O. D. Beyan, A. Iqbal, Y. Khan, A. Antoniades, J. A. Keane, P. Hasapis, C. Georgousopoulos, M. Ioannidi, S. Decker, and R. Sahay. Querying phenotype-genotype associations across multiple knowledge bases using semantic web technologies. In *BIBE*, pages 1–5. IEEE, 2013. (Cited on page 114.)

[17] C. Bizer and A. Schultz. The berlin sparql benchmark. *IJSWIS*, 5(2):1–24, 2009. (Cited on pages 3, 18, 21, 27, 123, 153, 155, 175, and 177.)

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. volume 13, pages 422–426, July 1970. (Cited on page 97.)

[19] D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema, 10. Februar 2004. W3C Recommendation. (Cited on page 10.)

[20] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. volume 60, pages 327–336, 1998. (Cited on pages 5, 97, and 99.)

[21] S. Capadisli, S. Auer, and A.-C. N. Ngomo. Linked sdmx data. *SWJ*, 2014. (Cited on page 113.)

[22] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. In *WWW*, pages 613–622, 2007. (Cited on page 117.)

[23] L. Chin, W. C. Hahn, G. Getz, and M. Meyerson. Making sense of cancer genomic data. *Genes & development*, 25(6):534–555, 2011. (Cited on page 127.)

[24] K. G. Clark, L. Feigenbaum, and E. Torres. SPARQL Protocol for RDF. World Wide Web Consortium, Recommendation REC-rdf-sparql-protocol-20080115, January 2008. (Cited on page 11.)

[25] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: A comparison of rdf benchmarks and real rdf datasets. In *SIGMOD*, 2011. (Cited on pages xix, 3, 153, 154, 160, and 175.)

[26] C. Dwork. Differential Privacy. In *ICALP (2)*, pages 1–12, 2006. (Cited on page 115.)

[27] O. Görlitz and S. Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *COLD at ISWC*, 2011. (Cited on pages 2, 5, 6, 17, 18, 20, 24, 29, 34, 39, 59, 65, 67, 74, 75, 81, 89, 91, 113, 118, 156, and 165.)

[28] O. Görlitz and S. Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *COLD, ISWC*, 2011. (Cited on pages 97 and 107.)

[29] O. Görlitz and S. Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *Proceedings of the 2nd International Workshop on Consuming Linked Data, Bonn, Germany*, 2011. (Cited on page 147.)

[30] O. Görlitz, M. Thimm, and S. Staab. Splodge: Systematic generation of sparql benchmark queries for linked open data. In *ISWC*. 2012. (Cited on pages 14, 23, 24, 155, 175, 176, and 179.)

[31] J. Grant and D. Beckett. RDF Test Cases, 10. Februar 2004. W3C Recommendation. (Cited on page 10.)

[32] Y. Guo and J. Heflin. LUBM: A benchmark for owl knowledge base systems. *JWS*, 2005. (Cited on pages 3, 21, 153, 155, and 175.)

[33] O. Görlitz and S. Staab. Federated data management and query optimization for linked open data. In A. Vakali and L. Jain, editors, *New Directions in Web Data Management 1*, volume 331 of *Studies in Computational Intelligence*, pages 109–137. Springer Berlin Heidelberg, 2011. (Cited on page 20.)

[34] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *WWW*, 2010. (Cited on pages xvii, xviii, 75, and 92.)

[35] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 411–420, 2010. (Cited on page 106.)

[36] O. Hartig. An overview on execution strategies for linked data queries. *Datenbank-Spektrum*, pages 1–11, 2013. (Cited on pages 20 and 28.)

[37] O. Hartig, C. Bizer, and J.-C. Freytag. Executing sparql queries over the web of linked data. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 293–309, 2009. (Cited on page 32.)

[38] A. Hasnain, R. Fox, S. Decker, and H. F. Deus. Cataloguing and linking life sciences lod cloud. In *OEDW at EKAW*, 2012. (Cited on pages 29, 30, and 31.)

[39] A. Hasnain, M. R. Kamdar, P. Hasapis, D. Zeginis, C. N. Warren Jr, et al. Linked Biomedical Dataspace: Lessons Learned integrating Data for Drug Discovery. In *International Semantic Web Conference (In-Use Track), October 2014*, 2014. (Cited on page 31.)

[40] P. Hayes and B. McBride. RDF Semantics, 10. Februar 2004. (Cited on page 10.)

[41] K. Hose and R. Schenkel. Towards benefit-based rdf source selection for sparql queries. In *SWIM*, page 2, 2012. (Cited on page 97.)

[42] T. J. Hudson, W. Anderson, A. Aretz, A. D. Barker, C. Bell, R. R. Bernabé, M. Bhan, F. Calvo, I. Eerola, D. S. Gerhard, et al. International network of cancer genome projects. *Nature*, 464(7291):993–998, 2010. (Cited on page 127.)

[43] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *ACM SIGMOD Record*, volume 20, pages 168–177. ACM, 1991. (Cited on page 55.)

[44] J. Jeong, L. Li, Y. Liu, K. Nephew, T. Huang, and C. Shen. An empirical bayes model for gene expression and methylation profiles in antiestrogen resistant breast cancer. *BMC medical genomics*, 3(1):55, 2010. (Cited on page 127.)

[45] U. Juergen, H. Aidan, P. Axel, and D. Stefan. Link traversal querying for a diverse web of data. *Semantic Web Journal*, 2013. (Cited on page 20.)

[46] E. Kamateri, E. Kalampokis, E. Tambouris, and K. Tarabanis. The linked medical data access control framework. *Journal of Biomedical Informatics*, 2014. (in press). (Cited on page 117.)

[47] M. Kamdar, A. Iqbal, M. Saleem, H. Deus, and S. Decker. Genomesnip: Fragmenting the genomic wheel to augment discovery in cancer research. In *Conference on Semantics in Healthcare and Life Sciences (CSHALS)*, 2014. (Cited on page v.)

[48] M. Kamdar, A. Iqbal, M. Saleem, H. Deus, and S. Decker. Genomesnip: Fragmenting the genomic wheel to augment discovery in cancer research. In *CSHALS*, 2014. (Cited on page 175.)

[49] M. R. Kamdar, D. Zeginis, A. Hasnain, S. Decker, and H. F. Deus. ReVeaLD: A user-driven domain-specific interactive search platform for biomedical research. *Journal of Biomedical Informatics*, 47(0):112 – 130, 2014. (Cited on page 31.)

[50] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi. Atlas: Storing, updating and querying rdf(s) data on top of dhts. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4), 2010. (Cited on pages 29 and 30.)

[51] J. Karlsson, O. Torreño, D. Ramet, G. Klambauer, M. Cano, and O. Trelles. Enabling large-scale bioinformatics data analysis with cloud computing. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 640–645. IEEE, 2012. (Cited on page 127.)

[52] Y. Khan, M. Saleem, A. Iqbal, M. Mehdi, A. Hogan, P. Hasapis, A.-C. Ngonga Ngomo , S. Decker, and R. Sahay. Safe: Policy aware sparql query federation over rdf data cubes. In *Semantic Web Applications and Tools for the Life Sciences (SWAT4LS)*, 2014. (Cited on pages vi, 6, 7, and 29.)

[53] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax, 10. Februar 2004. W3C Recommendation. (Cited on page 10.)

[54] G. Ladwig and T. Tran. Linked data query processing strategies. In *ISWC*, pages 453–469. 2010. (Cited on pages 20, 29, and 30.)

[55] G. Ladwig and T. Tran. Sihjoin: Querying remote and local linked data. In *The Semantic Web: Research and Applications*, volume 6643, pages 139–153. 2011. (Cited on pages 29 and 30.)

[56] A. Langegger, W. Wöß, and M. Blöchl. 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, 2008. (Cited on page 32.)

[57] S. Lynden, I. Kojima, A. Matono, and Y. Tanimura. Aderis: An adaptive query processor for joining federated sparql endpoints. In *OTM*, pages 808–817. 2011. (Cited on pages 2, 18, 29, 34, 35, 65, and 67.)

[58] F. Manola and E. Miller. RDF Primer, 10. Februar 2004. W3C Recommendation. (Cited on page 10.)

[59] S. Michel, M. Bender, P. Triantafillou, and G. Weikum. Iqn routing: Integrating quality and novelty in p2p querying and ranking. In *EDBT*, pages 149–166, 2006. (Cited on page 100.)

[60] G. Montoya, H. Skaf-Molli, P. Molli, and M.-E. Vidal. Fedra: Query processing for sparql federations with divergence. *arXiv preprint arXiv:1407.2899*, 2014. (Cited on page 32.)

[61] G. Montoya, M.-E. Vidal, and M. Acosta. A heuristic-based approach for planning federated sparql queries. In *COLD*, 2012. (Cited on pages 17, 20, 24, 35, 37, 38, 39, 45, 47, 50, 59, 65, 66, 74, 76, 91, 156, and 168.)

[62] G. Montoya, M.-E. Vidal, O. Corcho, E. Ruckhaus, and C. Buil-Aranda. Benchmarking federated sparql query engines: are existing testbeds enough? In *ISWC*, pages 313–324. 2012. (Cited on pages 5, 18, 20, 24, 44, 153, 155, 158, 164, and 168.)

[63] M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo. Dbpedia sparql benchmark - performance assessment with real queries on real data. In *International Semantic Web Conference*, pages 454–469, 2011. (Cited on pages 3, 18, 23, 153, 155, 175, and 179.)

[64] M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo. Dbpedia sparql benchmark: performance assessment with real queries on real data. In *Proceedings of the 10th international conference on The semantic web, Volume Part I*, ISWC'11, pages 454–469, 2011. (Cited on page 106.)

[65] A.-C. Ngonga Ngomo. On link discovery using a hybrid approach. *J. Data Semantics*, 1(4):203–217, 2012. (Cited on page 137.)

[66] A.-C. Ngonga Ngomo and S. Auer. LIMES - A time-efficient approach for large-scale link discovery on the web of data. In *IJCAI*, 2011. (Cited on pages 176 and 180.)

[67] A.-C. Ngonga Ngomo, S. Auer, J. Lehmann, and A. Zaveri. Introduction to linked data and its lifecycle on the web. In *Reasoning Web*, 2014. (Cited on page 1.)

[68] F. Picalausa and S. Vansummeren. What are real sparql queries like? In *SWIM*, 2011. (Cited on pages 3, 24, 153, 155, and 175.)

[69] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. (Cited on page 11.)

[70] B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. In *ESWC*, pages 524–538, 2008. (Cited on pages 2, 18, 20, 29, 34, 65, 67, 75, and 113.)

[71] B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. In *ESWC*, pages 524–538, 2008. (Cited on pages 97 and 107.)

[72] N. A. Rakhmawati, M. Saleem, S. Lalithsena, and S. Decker. Qfed: Query set for federated sparql query benchmark. In *The 16th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, 2014. (Cited on page vi.)

[73] N. A. Rakhmawati, J. Umbrich, M. Karnstedt, A. Hasnain, and M. Hausenblas. Querying over federated sparql endpoints - a state of the art survey. volume abs/1306.1723, 2013. (Cited on pages 17, 20, and 59.)

[74] Richard Cyganiak, Dave Reynolds, Jeni Tennison. The RDF Data Cube Vocabulary, January 2014. (Cited on page 115.)

[75] R. Sahay, D. Ntalaperas, E. Kamateri, P. Hasapis, O. D. Beyan, M. F. Strippoli, C. Demetriou, T. Gklarou-Stavropoulou, M. Brochhausen, K. A. Tarabanis, T. Bouras, D. Tian, A. Aristodimou, A. Antoniades, C. Georgousopoulos, M. Hauswirth, and S. Decker. An ontology for clinical trial data integration. In *IEEE International Conference on Systems, Man, and Cybernetics, Manchester, SMC 2013, United Kingdom, October 13-16, 2013*, pages 3244–3250. IEEE, 2013. (Cited on page 117.)

[76] M. Saleem, I. Ali, A. Hogan, Q. Mehmood, and A.-C. Ngonga Ngomo. Lsq: The linked sparql queries dataset. In *International Semantic Web Conference*. 2015. (Cited on page v.)

[77] M. Saleem, A. Hasnain, and A.-C. Ngonga Ngomo. Feasible: A featured-based sparql benchmarks generation framework. In *International Semantic Web Conference*, pages 561–576, 2013. (Cited on pages v, 2, 3, 5, 7, 18, 24, 29, 31, 35, 39, 44, 65, 74, 91, 156, and 175.)

[78] M. Saleem, A. Hasnain, and A.-C. Ngonga Ngomo. Bigrdfbench: A billion triples benchmark for sparql query federation. In *International Semantic Web Conference*. 2015. (Cited on pages v, 6, 7, 8, 10, 17, and 175.)

[79] M. Saleem, A. Hasnain, and A.-C. Ngonga Ngomo. Bigrdfbench: A billion triples benchmark for sparql query federation. In *International Semantic Web Conference*. 2015. (Cited on pages vi, 6, 7, 10, and 153.)

[80] M. Saleem, M. R. Kamdar, A. Iqbal, S. Sampath, H. F. Deus, and A.-C. N. Ngomo. Big linked cancer data: Integrating linked tcga and pubmed. *Web Semantics: Science, Services and Agents on the World Wide Web*, 27:34–41, 2014. (Cited on page 175.)

[81] M. Saleem, M. R. Kamdar, A. Iqbal, S. Sampath, H. F. Deus, and A.-C. Ngonga Ngomo. Big linked cancer data: Integrating linked tcga and pubmed. *Web Semantics: Science, Services and Agents on the World Wide Web*, 27:34–41, 2014. (Cited on page v.)

[82] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. N. Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *Semantic Web Journal*, 2014. (Cited on page 147.)

[83] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. N. Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *SWJ*, 2015. (Cited on pages xviii and 92.)

[84] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *Semantic Web Journal*, 2014. (Cited on pages v, 7, and 17.)

[85] M. Saleem, R. Maulik, I. Aftab, S. Shanmukha, H. Deus, and A.-C. Ngonga Ngomo. Fostering serendipity through big linked data. In *Semantic Web Challenge at International Semantic Web Conference*, pages 1–8, 2013. (Cited on page v.)

[86] M. Saleem and A.-C. N. Ngomo. Hibiscus: Hypergraph-based source selection for sparql endpoint federation. In *ESWC*, 2014. (Cited on page 118.)

[87] M. Saleem and A.-C. Ngonga Ngomo. HiBISCuS: Hypergraph-based source selection for sparql endpoint federation. In *Extended Semantic Web Conference*, pages 176–191. 2014. (Cited on pages v, 5, 6, 7, 10, 14, 65, 81, 153, 155, 156, 164, 165, 166, and 168.)

[88] M. Saleem and A.-C. Ngonga Ngomo. Quetsal: A query federation suite for sparql. In *International Semantic Web Conference*. 2015. (Cited on pages vi, 5, 6, 7, 10, 29, 31, and 81.)

[89] M. Saleem, S. S. Padmanabhuni, A.-C. N. Ngomo, J. S. Almeida, S. Decker, and H. F. Deus. Linked cancer genome atlas database. In *Proceedings of the 9th International Conference on Semantic Systems*, pages 129–134, 2013. (Cited on pages v, 1, 17, 20, and 159.)

[90] M. Saleem, S. S. Padmanabhuni, A.-C. Ngonga Ngomo , A. Iqbal, J. S. Almeida, S. Decker, and H. F. Deus. Topfed: Tcga tailored federated query processing and linking to lod. *Journal*

of Biomedical Semantics, 5(1):47, 2014. (Cited on pages v, 6, 7, 81, 127, 159, and 161.)

[91] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: a benchmark suite for federated semantic data query processing. In *ISWC*, 2011. (Cited on pages 2, 3, 66, 74, 82, 91, 153, 155, 159, and 175.)

[92] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: a benchmark suite for federated semantic data query processing. In *ISWC*, pages 585–600. 2011. (Cited on pages 18, 24, and 38.)

[93] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Spˆ 2bench: a sparql performance benchmark. In *ICDE*, pages 222–233, 2009. (Cited on pages 3, 18, 153, 155, and 175.)

[94] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *ISWC*, pages 601–616. 2011. (Cited on pages 2, 5, 6, 17, 18, 20, 21, 24, 29, 31, 34, 39, 59, 65, 67, 74, 75, 81, 89, 91, 113, 118, 122, 156, 165, and 175.)

[95] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *ISWC*, Lecture Notes in Computer Science, pages 601–616. 2011. (Cited on pages 97, 107, 128, and 149.)

[96] A. Schwarte, P. Haase, M. Schmidt, K. Hose, and R. Schenkel. An experience report of large scale federations. *CoRR*, abs/1210.5403, 2012. (Cited on page 20.)

[97] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979. (Cited on pages 34 and 170.)

[98] K. D. Siegmund. Statistical approaches for the analysis of dna methylation microarray data. *Human genetics*, 129(6):585–595, 2011. (Cited on page 127.)

[99] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres. Comparing data summaries for processing live queries over linked data. *World Wide Web*, 14(5-6):495–544, 2011. (Cited on page 20.)

[100] X. Wang, T. Tiropanis, and H. C. Davis. Lhd: Optimising linked data query processing using parallelisation. In *LDOW at WWW*, 2013. (Cited on pages 2, 17, 18, 20, 21, 29, 34, 65, 67, 113, and 118.)

[101] Wikipedia. SPARQL — Wikipedia, The Free Encyclopedia, 2013. [Online; accessed 31-March-2013]. (Cited on page 11.)

[102] H. Wu, T. Fujiwara, Y. Yamamoto, J. Bolleman, and A. Yamaguchi. Biobenchmark toyama 2012: an evaluation of the performance of triple stores on biological data. *JBMS*, 2014. (Cited on pages 3, 153, and 155.)