Doctoral Thesis

Efficient Path Query Approaches Over Distributed Linked Data

Qaiser Mehmood

March 15, 2021

Supervisor
Prof. Mathieu dÁquin

Co-supervisor
Dr. Ratnesh Sahay

External Examiner
Prof. Pascal Molli

External Examiner
Prof. Thomas Riechert

Internal Examiner
Prof. John Breslin

Insight Centre for Data Analytics
College of Engineering and Informatics, National University of Ireland, Galway

ABSTRACT

In the past decade, graphs have become the defacto data model in some of the popular application domains such as bioinformatics, social, and road networks. One of the most used variants of the graph model is the labeled graph where vertices and edges are given names. In the context of Semantic Web technologies, this graph representation is called Linked Data, expressed using RDF (Resource Description Format), where entities are given their names with Uniform Resource Identifiers (URIs) which uniquely identify each entity. A plethora of such information, corresponding to various domains, exists over the Web and is exposed as knowledge bases. Some of the prominent examples of such knowledge bases or datasets are YAGO, DBPedia and Bio2RDF. Things or entities described within these datasets can be interconnected within a single graph or these connections may lead to different datasets.

This large collection of interlinked and distributed data over the web has created new challenges in the context of data management, data integration and knowledge discovery. One of the key challenges is querying and analysing relevant information from such large-scale knowledge bases. For instance, in the context of graph analysis, navigational queries are at the basis of core tasks and processes. Several approaches have been proposed to perform path queries, with some supporting declarative queries, while others are based on imperative constructs. Moreover, some approaches work in a distributed manner while others in a central way. For instance, SPARQL1.1, which is a declarative query language, introduced a feature called Property Paths (PPs). Using PPs, a user can check the existence of paths between two entities within a single graph. However, this variant is not very expressive, does not capture some of the important properties of linked data, and has performance issues when dealing with large-scale data.

In order to exploit the full potential of linked data, the standard declarative query language of SPARQL needs (1) to be equipped with expressive features to capture some important graph analytical tasks and (2) to perform better even for large-scale graphs. We propose an extension to SPARQL1.1 property paths which we believe complements the current version of PPs with analytical features in a more expressive and robust manner. The algorithm proposed in our extension is a relatively straightforward solution but performs efficiently compared to tailored solutions in the literature. Our solution allows users to compute k shortest paths matching a property expression between two nodes.

While the constant growth of linked data on the web in various domains has led to a challenging environment for data processing, it has also justified the need for novel approaches to handle and analyse such big data. Ecosystems such as Hadoop or Pregel have become defacto standards for such tasks and are capable of navigating the path queries in a distributed manner. However, these systems are developed for customised data distribution within a controlled environment, and in the context of Linked Data such systems are not considered purely heterogeneous. In our second major contribution in this thesis, considering the limitations and management efforts to handle homogeneous systems, we propose different solutions which we believe are easy to adopt and scalable. They do not require complicated management settings when the data is either customized in a controlled environment or even if the data is accessible publicly in a heterogeneous environment.

Our first implementation towards distributed path query processing is FedS, which we propose as a P2P solution. However, our solution in contrast with the core P2P architecture is of a *hybrid* type. Generally, queries in P2P networks are blindly forwarded from node to node, and also a loose guarantee of resource discovery in P2P network can make it possible not to find a resource node although it does exist. We introduced the concept of source selection within the P2P network, which reduces the number of query requests by selecting the relevant source nodes.

Our second implementation for distributed architecture is QPPDS, which is an *index-based* path query engine with the concept of finding paths within a heterogeneous environment. We proposed a source selection mechanism based on a pre-computed index. We also provide a dataset, exposed as a SPARQL endpoint, for the paths calculated by QPPDS. This dataset contains the statistical and provenance information for those paths. We provide a comprehensive evaluation of this approach on real-world life sciences data with different execution strategies.

Our third implementation is a *cache-based* and *index-free* distributed path query engine called DpcLD, which computes the paths distributed within a homogeneous environment. We provide extensive evaluations on both *synthetic* and *real-world* data. We evaluate DpcLD not only against the distributed approaches but also do a comparison with the centralized approach and show *trade-offs* in different contexts.

In this thesis, we proposed different approaches to address the pathfinding task, especially in a distributed environment. From the individual contributions described above, the overall conclusion of this work is that graph management and distribution have a strong impact on the efficiency and accuracy of pathfinding, and therefore that different solutions are required depending on their characteristics.

CONTENTS

DECLARATION

I declare that this thesis, titled "*Efficient Path Query Approaches Over Distributed Linked Data*", is composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification.

Galway, March 15, 2021

Qaiser Mehmood

DEDICATION

This dissertation is dedicated to my respected *Parents, Sisters, Wife, Arham and Simrah*. This thesis is also equally dedicated to my cousin *Yousaf*: you have been an inspiration to me.

ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my advisor Prof. Mathieu dÁquin for his guidance and patience. I could not have imagined having a better advisor. His comments, feedback, and proofreading, both in research papers and thesis, led me to improve the quality of the research work I conducted during my PhD. I would like to also thank Dr. Ratnesh and Prof. Dietrich Rebholz-Schumann for their support in the earlier stage of this work.

Many thanks go to my graduate research committee: Prof. John Breslin, Dr. Ali Intizar and Dr. Edward Curry.

I am also thankful to Prof. Axel Polleres and Vadim Savenkov for a collaborative work during my research visit to the Institute of Data, Process and Knowledge Management, Vienna University of Economics and Business, Austria.

I am also thankful to Dr. Muhammad Saleem from Agile Knowledge Engineering and Semantic Web (AKSW), Germany who worked with me as a coauthor of my research publications. His research feedback always helped me.

The period I spent in the Data Science Institute and the Insight Centre for Data Analytics, Galway was a great experience, and I am happy to be a part of the history of the research institute. I want to thank all my colleagues, and the administrative and technical staff at Insight Galway.

I am grateful to my parents for their countless prayers and support throughout my life. I thank my wife Khansa for her patience, and sacrificial care for me and the beautiful kids God has gifted us. I dedicate this thesis to my children Arham and Simrah whose smile make me the happiest man in the world and when I think of how I love you both, there are no words to explain. I love you because you are you, it's that simple and plain.

Finally, I am humbled by the blessings of the Creator and Sustainer of the universe, Allah s.w.t. Indeed, it is He who grants us what we do not deserve and none is worthy of praise except Him.

1 INTRODUCTION

Graph has a history of being used as a basis for exploring and solving complex problems in mathematics, resulting in many applications in the area of computer science. Starting with the solution of the "Seven Bridges of Königsberg" problem back in 1736 by the mathematician Leonhard Euler, graphs have become core to many applications. A graph is a representation of information, modeled as vertices and edges, to model the relationships among different objects. Graphs have become the *de facto* data model in some of the popular application domains such as bioinformatics, social networks, knowledge graphs, natural science, etc. In real-life scenarios, graphs are modeled according to the requirements of the applications. One of the most used variants of the graph model is labeled graph, where vertices and edges are given names (labels). This notion of labeled graphs is in particular at the core of what is called the "Semantic Web" or "Linked Data", and often referred to as the "Web of Data" [BHB11a; HB11]. The publication of such graph-based knowledge is based on standard Web technologies: Uniform Resource Identifiers (URIs) [BFM+98], Hypertext Transfer Protocol(HTTP) [FGM+99], and the Resource Description Framework (RDF) [Kly04] (formal definitions of relevant concepts are included in Chapter 2 of this thesis). With the advent of the World Wide Web (WWW) [Ber92; BDM+94] the information disseminated as Linked Data has been increasingly used by the research and enterprise community. In recent years, an exponential growth[1] of scientific and engineering data exposed on the Linked Open Data (LOD) cloud[2] (see Figure 2.3 of Chapter 2) has been witnessed. More and more structured content [BHB11b; JCB11; MP12; MB12] is appearing every day on the Web and is made publicly available in the form of documents or datasets.

An example of graph-based knowledge –i.e., Linked Data– is the Life Sciences data that exists over the LOD Web. This includes BIO2RDF [BNT+08], BIOPORTAL [NSW+09] or NBDC [KKH+18]. The data provided by these sources are modeled as resources (vertices) and properties (edges) and have semantically rich information about genes, diseases, and drugs. Other general-knowledge datasets, for instance, DBPedia [ABK+07] or YAGO [SKW07] also contain useful common-knowledge facts, for example about "Albert_Einstein", "birthPlace", "Ulm".

This large collection of interlinked and distributed data over the Web has created new challenges in data management, knowledge discovery, interoperability, and integration. These challenges have led academia and industry to pay more attention to efficiently storing, traversing and querying these enormous, heterogeneous and information-rich knowledge graphs.

For instance, in the context of graph analysis, in the Life Sciences domain one of the key challenges in cancer genomics is to discover gene-disease-drug *associations* or *paths*. This corresponds to understanding which gene is affected by what disease and how this disease can be treated with what drug. At the data level, such associations are indispensable [WDS+12] and provide insight into the drug development process –tailored specifically for an individual patient (or

1 see LOD stats: http://lodstats.aksw.org/
2 LOD Cloud: https://lod-cloud.net

a group of patients)– targeting prevention, diagnosis and treatment of diseases [SR13a]. However, in *real-world* data, this information may be centralized in large-scale graphs or distributed over different datasets. Therefore, searching these useful *relationships* may lead to the following problems:

(i) For large-scale graphs, data management and querying can be extremely resource-consuming.

(ii) For distributed data, finding paths between two entities distributed across various data sources can be very complex.

These problems can be addressed by introducing efficient path query solutions both for large-scale and distributed graphs. Thus, the aim of this thesis is to study efficient path query processing over large-scale and distributed graph datasets. The first part of this describes our work to efficiently find paths in large-scale graphs within a centralized environment. The second part of this thesis discusses pathfinding in distributed environments using the same query template we propose in the first part.

1.1 LINKED DATA AND NAVIGATIONAL QUERY

There are many options for extracting information from Linked Data. However, an expressive approach is to execute a declarative query. In Chapter 3 we present a feature based comparison of different proposals for RDF query languages. However, in the rest of our work, we focused only on the SPARQL language. This is because the SPARQL query language [HSP13] is a W3C [PS+17] standard and a prominent query language used for querying linked data.

SPARQL is accessed using the SPARQL protocol [FWC+13]. Therefore any web service that supports both SPARQL and its protocol is called a SPARQL endpoint. SPARQL queries, supported by an endpoint, are evaluated based on sub-graph matching.

EXAMPLE 1.1. The SPARQL query in Listing 1 is an example of a simple query, which returns a list of people who know *directly* http://dbpedia.org/resource/alice. In the path context, this is a one *hop* relation.

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT * WHERE {?people foaf:knows dbpedia:alice}
```

Listing 1: SPARQL query.

The query in Listing 1 cannot return any *indirect* relation. For example, if we want to find multi-hops of the foaf:knows relation, we have to explicitly define the paths using multiple variables, and the SPARQL query could be very complex. To address this limitation, a recent extension of SPARQL introduced a pathfinding variant called Property Paths[3] (PPs) (a.k.a. *regular path queries [CDL+00]*).

3 SPARQL Property Paths: https://www.w3.org/TR/sparql11-query/#propertypaths

EXAMPLE 1.2. In contrast to the SPARQL query of Listing 1, the following PP query in Listing 2 evaluates the foaf:knows relation *implicitly* through its *transitive closure*, i.e. this query evaluates the list of all people who know http://dbpedia.org/resource/alice by one or more-than-one path length.

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT * WHERE {?people foaf:knows+ dbpedia:alice}
```

Listing 2: A simple SPARQL Property Path query.

SPARQL1.1. Property Path queries allow a user to route through between two nodes (i.e., *source* and *target*) within a graph. A user can evaluate the path query based on different *regular expressions* described in Section 2.3.2 of Chapter 2 of this thesis. However, PPs miss the great potential of Linked Data and still remain largely unusable for the most typical graph query tasks. For example, PPs cannot be used to enumerate the paths, hence, missing important features for graph analysis such as obtaining complete paths, shortest paths, or k most promising connections. The following example in Listing 3 illustrates those limitations.

EXAMPLE 1.3. Consider the following query where we want to check if http://dbpedia.org/resource/bob knows http://dbpedia.org/resource/alice through one or more arbitrary routes. This query does not generate the answer because, while it can say if the path exist, it will not enumerate the paths or find the shortest paths.

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT * WHERE {dbpedia:bob foaf:knows+ dbpedia:alice}
```

Listing 3: Limitation example of a SPARQL Property Paths query.

Until now, we discussed the centralized querying of Property Paths. However, as mentioned previously, there are many approaches to querying RDF data, and we also show a feature-based comparison of different query languages in Table 3.1 of Chapter 3. Some of these languages in the table have been tested or are able to carry out distributed query processing. Examples are SPARQL [HSP13] and RDFPath [PSH+11]. However, those approaches have some limitations. For example, in the context of SAPRQL1.1., although the SERVICE clause has been introduced in the language to query distributed or federated datasets, however, in the context of PPs it is not feasible to query the data in a transitive way because of the complexity of query. Moreover, using SERVICE a query may return uncompleted results, as we do not know a priori knowledge about the path structure, and there is no work exist for the source selection in the context of PP queries.

Recently, in the research around link traversal [UHP+15], the concept of live SPARQL query processing over the Web has been presented. Hartig [HP17] proposed a way to compute property paths over the Web of Linked Data. The work presented in [HP17] has some limitations (which we discuss in Section 3.3.6 of Chapter 3), and is also out of the scope of this thesis. Other

than SPARQL queries, some other declarative queries, for instance, RDFpath [PSH+11] or Horton+ [SEH+13], are also available to query paths over distributed data sources. Some work also has been done where declarative queries are not used, but procedural programming or some other techniques [WWZ16a; DE16] are used to query the paths.

However, we noticed that there is a shortage of work done in the context of path queries over federated and distributed Linked Data graphs. For example, to the best of our knowledge, the existing implementations are either Hadoop-based [SKR+10], Pregel-based [MAB+10] or partition-based [SEH+13]. However, these models make the assumption that they are applied within homogeneous distributed systems. We briefly explain the difference between homogeneous and heterogeneous systems in Chapter 2. We also highlight their pros and cons in Chapter 3.
In summary, the existing approaches on distributed path queries focus on different aspects such as querying centralised data, query performance, sophisticated data partitioning techniques, and customised data models. This thesis fills the gaps in existing work by focusing on PP querying over distributed, connected graph data sources.

1.2 PROBLEM STATEMENT

As we have discussed previously, the plethora of interlinked data has created new challenges for data management, knowledge discovery, etc. One of the key challenges is to query these heterogeneous large-scale and distributed Linked Data graphs. The current SPARQL 1.1. property path queries are not sufficient to fully address these challenges. There is no mechanism in SPARQL PP querying, for instance, to enumerate and retrieve the complete paths, k prominent path, or shortest paths. In addition, the performance of PP querying is low [ACP12a; DGH+14]. Although some early approaches [KJ07; AMS07; KRU15; GN11] have proposed extensions for SPARQL PPs, they aimed either at improving performance or at enhancing the query features. None of them targeted both limitations of SPARQL PPs. Therefore, they did not gain much attention in the Semantic Web community.

These research gaps in prior approaches and the limitations in the current SPARQL 1.1. Property Paths (PPs) motivate us to address these problems. The first part of this dissertation therefore looks first at extending the SPARQL 1.1. property path features, and the second part looks at enabling such PP querying over distributed, connected graphs. Indeed, Linked Data by nature is distributed and heterogeneous. Thus, it seems natural to adapt such solutions efficiently and in a scalable manner so that they can work within a distributed computing paradigm.

It is worth mentioning that the second part of this thesis focuses on the mechanisms for distributed PP querying, including source selection and distributed path computation, and is not particularly related to the query semantics or declarative queries. We use our proposed path query only as a supportive language to construct the queries. Thus, even though we put some emphasis on SPARQL in the first part, our distributed solutions are independent of the query language used.
In summary, the problem statement is based on the following findings, in terms of limitations, in the current state of the art:

- Many state-of-the-art systems, which support path query languages, perform badly on large-scale graphs.

- Some path query languages such as SPARQL are not able to enumerate the paths but can only evaluate the path's existence and miss some important features, e.g. K-shortest paths.
- Linked data by nature is distributed. Therefore, most of the approaches that work in a centralized way (i.e., only on single graphs) would require, to be applicable in this context, to merge all sources together, which can create issues of scale and in keeping the data up-to-date.
- The current systems that provide the facility to find paths in a distributed environment have the following challenges:

  i) Most of the systems are homogeneous distributed systems. They work in a controlled environment where every distributed source or database must have the same software and configuration. Furthermore, the partition and distribution of data must follow certain procedures or constraints, which is incompatible with Linked Data sources.

  ii) Most of the homogeneous systems require procedural programming to construct a path query because they do not support declarative queries for pathfinding.

Consequently, there are research gaps and questions which need attention to address these limitations in large-scale graphs both for centralized and distributed graphs.

1.3 RESEARCH QUESTIONS

To address the limitations and problems mentioned above, our work attempts to answer the following questions:

Q1. How to extend SPARQL1.1. Property Paths query language so to efficiently address its limitations?

Q2. How path queries can be evaluated in distributed environments taking into consideration different aspects of the data distribution?

  i What alternative solutions to homogeneous distributed approaches can be adopted, where data partitioning and distribution mechanisms do not require any constraint or procedure to follow, and the solutions should be simple to configure?

  ii How can distributed path computation be efficient in heterogeneous and federated environment?

On the one hand, by answering those research questions, our proposed solutions would not only enhance the navigational and analytical capability of SPARQL1.1. Property Paths, but also increase the query performance which is currently a bottleneck in current PP querying. On the other hand, our distributed solutions (for homogeneous computing paradigm) should be efficient, easy to plugin, and not require sophisticated data sharing mechanisms or strict configurations. Our federated path computation approach should make it possible to query paths in heterogeneous environments.

1.4 CONTRIBUTIONS AND THESIS STRUCTURE

In this thesis, we cover different aspects of data management and querying in (i) centralized systems and (ii) distributed systems.

The first part considers centralized approaches, where our contribution is as follows:

1. To answer the first research question **Q1**, we proposed an extension to the current SPARQL1.1. property path queries, where we introduced a customised function that supports PPs semantics. Thus, instead of just confirming the existence of path connections, a user can find the *K* shortest paths efficiently in large-scale graphs. This work was published in 13th International Conference on Semantic Systems (SEMANTiCS 2017) [SMU+17].

In the context of distributed computing paradigm, we contributed and proposed solutions based on data distributions and querying techniques. For homogeneous distributed computing paradigms, we propose a *peer-to-peer* and *index-free* path query engine. For heterogeneous distributed computing, we propose an *index-based* path query engine.

2. To answer the research question **Q2**, the first distributed approach we proposed is FedS. This approach is a peer-to-peer (P2P) architecture. It is important to note that P2P architectures are considered in the taxonomy of homogeneous distributed systems. The problem and drawback of a P2P architectures is flooding of messages to each node for a given query request. Furthermore, the data partitioning and distribution must follow a custom procedure. In contrast to the standard P2P architecture, we proposed an architecture where there is no need for custom data partitioning and distribution. Instead, each node can have its own independent data. Moreover, in this approach, we introduced a source selection mechanism which solved the problem of message flooding. This work was published in 20th International Conference on Big Data Analytics and Knowledge Discovery (DaWaK 2018) [MJR+18] .

3. To address the research question **Q2**, we propose a federated pathfinding approach. Unlike the previous approach (FedS), QPPDs is an indexed-based data source selection and path query federation engine. As Linked Data by nature is heterogeneous, distributed and most of the data is exposed as SPARQL endpoints, where different databases provide path navigation functionalities, QPPDs brings back the resulting (partial or full) paths from those federated repositories and constructs the complete paths for a given request. This work was publised in journal (IEEE ACCESS 2019) [MSS+19]

4. Also addressing the research question **Q2**, we propose a cache-assisted and index-free distributed path query engine (DpcLD) based on a shared-algorithm. This shared-algorithm is deployed over different SPARQL endpoints. The engine communicates with the shared-algorithm and retrieves the partial answers from remote endpoints. In contrast to QPPDs, this architecture also lies in the taxonomy of homogeneous systems. The difference between this approach and, for example, Hadoop-based MapReduce systems is that our approach is easy to adopt, includes a source selection mechanism, and therefore avoids message flooding. These Hadoop-based systems are vertex-centric where each vertex sends a message to its connected vertices. Therefore, their communication cost may increase if the graphs are dense. Moreover, our approach does not require complex configurations or

customised graph partitioning and distribution strategies. This work is under-review in journal.

The organisation of the reminder of this thesis is as follows:

Chapter 2 briefly introduces the relevant concepts, terms, definitions, and notions that are necessary to understand the rest of the thesis.

Chapter 3 highlights the state of the art and related work. We also compare different approaches with our proposed solutions. In particular, we discuss distributed architectures as well as their advantages and disadvantages.

Chapter 4 addresses our first research question: Path query processing for large-scale graphs within a centralized environment. In this chapter, we propose an extension to the current SPARQL1.1. Property Paths and address the limitations in current version of path queries.

Chapter 5 presents an unstructured P2P-like architecture. It addresses the current problems (e.g., message flooding) in standard P2P architectures.

Chapter 6 presents our index-based federated path query engine that works in heterogeneous environments.

Chapter 7 presents our index-free, cache-assisted distributed path query engine that works within the homogeneous distributed environment and communicates with a shared-algorithm to compute the distributed paths.

Chapter 8 finally concludes the overall work we proposed in this dissertation, and provides some possible future directions.

2 BACKGROUND & PRELIMINARIES

To understand the content of this thesis, this chapter introduces the concepts, formal definitions, and topics relevant to other chapters. Since Linked Data is based on a representation of graphs, we start by introducing basic graph theory concepts. We then briefly explain concepts in graph data management, the Semantic Web and Linked Data. We give an overview of Linked Open Data, storage and querying in particular with SPARQL and property paths. We also discuss the foundations of path queries which were adopted in property paths. We briefly discuss the distributed processing of Linked Data, and highlight the difference between homogeneous and heterogeneous distributed computing paradigms.

2.1 GRAPHS

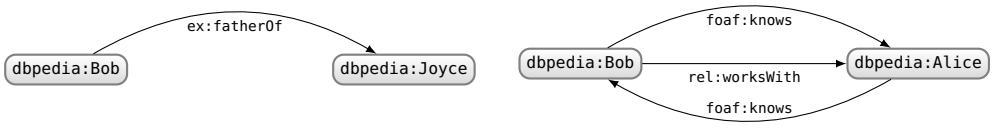
To capture the relationships between objects, graphs provide a versatile, flexible and simple data models. Real-world data in many domains, e.g., bioinformatics or social networks can therefore be modeled in graphs. A well explained overview of definitions and concepts of graph theory is available in the book written by Reinhard Deistal [Die12], where a graph is defined as follows:

Definition 1 *A **Graph** G denoted by G=(V,E) is a pair of sets, where V is a set of vertices and E is a set of edges which satisfies* $E \subseteq V \times V$

We can distinguish two types of graphs based on the types of edges used: *un-directed* and *directed* graphs.

Definition 2 *A **Directed Graph** G denoted by G=(V,E) is a pair of sets, where V is a set of ordered pairs of vertices and E is a set of directed edges satisfying* $E \subseteq V \times V$.

In directed graphs, (i) a *simple graph* is that which does not have cycles or multiple edges between two distinct vertices i.e., no two edges connect the same pair of vertices, (ii) a *multigraph* is one with multiple edges between two distinct vertices. Figure 2.1((a) and (b)), represent simple and multigraph examples respectively.



(a) Simple graph example

(b) Multigraph example

Figure 2.1: Two notions of directed (labelled) graphs

2.1.1 Graph Properties

The characteristics of a graph can be termed as its properties. In this thesis, we do not cover all the properties but only those which are relevant to our work. Detailed definitions of graph properties can be found in Reinhard Deistal's book [Die12].

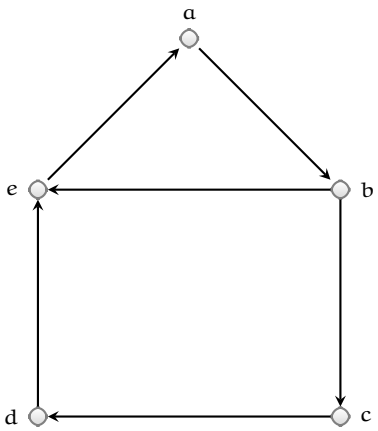


Figure 2.2: A directed graph to explain properties.

CARDINALITY

- the **Size** of a graph $G = (V, E)$ is the number of edges, denoted by $|E|$
- the **Order** of G is the number of vertices, denoted by $|V|$

EXAMPLE 2.1. The cardinality of the graph shown in Figure 2.2, is 6 in size, and 5 in order.

NEIGHBORHOODS

- the **Predecessors** of a vertex v is a set of vertices $u \in V$, denoted by $predecessor(v)$, in a directed graph $G = (V, E)$, that come just before the vertex v, such that $(u, v) \in E$. That is, $predessor(v) = \{u | (u, v) \in E\}$.
- the **Successors** of a vertex v is a set of vertices $u \in V$, denoted by $successor(v)$, in a directed graph $G = (V, E)$, that come immediately after the vertex v, such that $(v, u) \in E$. That is, $successor(v) = \{u | (v, u) \in E\}$.
- the **Neighbors** of a vertex v is the union of the two previous sets i.e., $predecessor(v)$ and $successor(v)$. That is, $neighbor(v) = predecessor(v) \cup successor(v)$

EXAMPLE 2.2. In Figure 2.2, graph shows the $succ(b) = \{e, c\}, predec(b) = \{a\}, neigh(b) = \{a, e, c\}$

CONNECTEDNESS Herein, we discuss the basic concepts of the graph connectedness that help us in understanding the topology of the graph $G = (v, E)$.

- a **Path** P is a chain of edges connecting two vertices (e.g., s, t) within a graph $G = (V, E)$, where s corresponds to the $source$ and t to $target$. Thus, a path of finite ordered set of edges can be written as:

$$[P]_{s,t} = ((u_0, u_1), (u_1, u_2), (u_2, u_3), ..., (u_{k-1}, u_k)) \mid s := u_0 \text{ and } t := u_k$$

a path can also be written as a sequence of vertices, as:

$$[P]_{s,t} = (u_0, u_1, u_2, u_3, ..., u_k) \mid s := u_0 \text{ and } t := u_k$$

here k represents the sequence number of a vertex u. A path p can be considered as a sub-graph of G, when a path is formulated as $P(V_p, E_p)$. Thus, V_p is a set of $\{u_0, u_1, u_2, u_3, ..., u_k\}$ and E_p is a set of $\{(u_0, u_1), (u_1, u_2), (u_2, u_3), ..., (u_{k-1}, u_k)\} \mid s := u_0$ and $t := u_k$.

The *path length* P^l is the number of edges involved to connect s and t. A path with minimum length (i.e., $P^l = 1$) is called a *direct* path. An *indirect* path is where $P^l > 1$ connected nodes are involved. If there is no repeated vertex in the path, that path is called *simple*.

The set of calculated paths, denoted by $\mathcal{P}(s,t)$, for a given source s and target t is the set of all the unique paths connecting s and t.

Paths in $\mathcal{P}(s,t)$ are unique and distinct, i.e. for any P_1 and P_2 of $\mathcal{P}(s,t)$ $P_1 \not\subseteq P_2$ and $P_2 \not\subseteq P_1$ are verified. The two paths P_1 and P_2 are said to be *edge-disjoint* i.e., $P_1 \cap P_2$ if they do not share any common edge. Likewise if P_1 and P_2 do not share a common vertex, they are said to be *vertex-disjoint*.

Given a set of paths $\mathcal{P}(s,t)$, the path P with the shortest length P^l in that set is called the *shortest* path.

- **Reachability** In a graph $G = (V, E)$, if there exists at least one path between s and t we say that s and t are reachable and we denote it by $s \rightsquigarrow t$. Reachability can be expressed as a boolean value i.e.,:

$$s \rightsquigarrow t = \begin{cases} \text{true}, & \text{if } \mathcal{P}s,t \neq \emptyset \\ \text{false}, & \text{otherwise} \end{cases}$$

EXAMPLE 2.3. In Figure 2.2, considering source a and target e: (a) the source and target are *reachable*, that is, $a \rightsquigarrow e$ is true; (b) paths between source and target are: $P = \{a, b, e\}, \{a, b, c, d, e\}$, (c) path lengths are $\{2, 4\}$, and (d) the shortest path is $\{a, b, e\}$.

2.1.2 Graph Data Management

This section provides background on graph data models and query languages commonly used in practice. We also discuss relevant state-of-the-art graph database systems that are commonly used for graph storage and querying purposes.

GRAPH DATA MODELS For many application domains graph data models are intrinsic. Here, we discuss some common graph data models used in many domains. A comprehensive description of the graph data models is available in the survey paper [AG08].

- **XML-based Structured Model:** Extensible Markup Language (XML) basically is a *tree* structured data model that represents tree based semi-structured, directed (see Definition 2) and acyclic graph data.

- **RDF-based models:** The Resource Description Framework (RDF) [Kly04] is a W3C recommendation and an example of directed labeled multi-graph data model. It is a widely used data format in the Semantic Web community. In this thesis, we used RDF as graph data model for our evaluations. Listing 2.1 shows an example of data represented using the RDF model. We briefly discuss the Linked Data, Semantic Web, and RDF terms in section 2.2. Examples of some large and widely used knowledge graphs modeled as directed labeled multi-graphs in RDF are: DBpedia [ABK+07], YAGO [SKW07], Bio2RDF [BNT+08], BioPortal [NSW+09]. Note that other knowledge graphs, such as the Google Knowledge Graph [Sin12], also use a directed labelled multigraph data model even if not natively in RDF.

- **Directed Labeled Property Graph Model:** Those graph models are an extended form of labeled graphs, with additional properties or attributes associated with both vertices and edges. The attributes are key-value pairs which define the properties of individual vertices and edges. The popular database (i.e., Neo4j [Neo]) uses property graphs as an underlying data format. There are many use cases of real-world data, for example from sensor networks and social networks, that are modeled using property graphs.

GRAPH QUERY MODELS Querying labeled graphs is one of the most important mechanism to extract information from data represented in such data models. In this section, we highlight some of the query models and supported languages for them. A brief discussion on query models and languages can be found in [AG08].

- **Pattern Matching Queries:** To query the graph, a popular model is the one of pattern matching queries. This type of queries deals with the subgraphs of the targeted graph, where a triple pattern $< u, e, v >$ (u and v being vertices or variables, and e being an edge label or a variable) of the query is matched against edges within the graph. We explain triples and pattern matching queries, including SPARQL queries [HSP13], in sections 2.2.2 and 2.3 respectively.

- **Connectivity Queries:** A fundamental paradigm of graph applications is the connectivity query. Different notions of queries such as *reachability*, *regular path*, k *path*, and *shortest path* queries are categorised as connectivity queries. An example of connectivity query is SPARQL 1.1. Property Paths [Sea] (see an example in Section 1.1). This thesis focuses only on this type of query, and we briefly explain path queries in Section 2.3.

Like SPARQL, there are many other graph query languages provided by different graph databases, which are more or less widely adopted, such as Cypher [Neo] and Gremlin [Rod15].

2.2 SEMANTIC WEB & LINKED DATA

In this section we briefly describe the notions of the World Wide Web, Web of Data (i.e., Linked Data or Semantic Web) and tools and technologies used for storage and querying purposes in that context.

2.2.1 The World Wide Web (WWW)

The advent of the World Wide Web (or simply the Web) by Tim Berners-Lee, made it possible for computers or machines to connect and share resources over the network via the Internet [MS00]. Documents published and located on different places can be accessed, using their globally assigned unique address (i.e., a Uniform Resource Locator, URL), over the Hypertext Transfer Protocol (HTTP). Since the emergence of the Web, the adaptation of database techniques for retrieving information has attracted much research interests. Thus, the popularity of the Web is considered as a *"primary vehicle for the dissemination of information"* [DAA05]. A set of best practices for publishing and connecting semi/structured data on the Web has emerged. Documents commonly written in Hypertext Markup Language (HTML) are know as html pages. There is another form of publication of data on the Web which has gained tremendous attraction in recent years, and that is referred to as *"the Semantic Web"* or *"Linked Data"*.

2.2.2 The Semantic Web

Traditionally, the approaches used for dissemination and consumption of information on the Web were human centric. However, with the advent of the Semantic Web, a concept put forward by Tim Berners-Lee [BHL01], it became a standard principle to share, find, and reuse information which is not only understandable by humans, but also for the machines to understand and process. Thus, the main motivation behind this concept was bootstrapping web intelligent machines.

The concept of the Semantic Web can be considered as *"a variation, or perhaps more realistically, an augmentation of the current Web such that it is more amenable to machine processing, and such that software agents can accomplish many of the tasks that users must currently perform manually"* [Hog11].

Tim Berners-Lee outlined the conceptual baseline for the Semantic Web as:

"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"

The research work presented in this document is based on Semantic Web technologies. Thus, it is important to understand the building blocks – resources, URIs, RDF, Linked Data – of the Semantic Web, as briefly discussed in the following paragraphs.

UNIFORM RESOURCE IDENTIFIER (URI) Universal Resource Identifiers (URIs) are a building block of data represented as Linked Data over the Web. They represent a formal way to refer to Web resources unambiguously distinguishing any resource on a global level. The generic syntax for URIs is defined in RFC 3986 [1]. URIs can be resolvable, over the Web (or more precisely the HTTP protocol), for the resources they refer to. The unique identification of resources enable them to interact with their representations on the Web.

As per the W3C recommendations for so-called "Cool URIs for the Semantic Web", it is important to understand that URIs make a distinction between a thing (which may exist outside the web) and a web document describing the thing" [2]. In Semantic Web perspective, dereferencable URIs of resources point to RDF documents which contain additional information about these resources.

1 RFC3986: https://tools.ietf.org/html/rfc3986
2 Cool URIs: https://www.w3.org/TR/cooluris/#distinguishing

INTERNATIONALIZED RESOURCE IDENTIFIERS (IRI) Both URI and IRI are the glue that holds the Web together. IRIs allow content developer and users to identify resource in their own languages [3]. IRI specification provides an important and critical reference to many W3C specifications such as XML, RDF, SVG, XHTML for identifier that support international characters.

According to IRI specification, every URI is also an IRI, and users do not need to do anything extra to find their resource on the Web. The generic syntax for IRIs is defined in RFC 3987 [4].

RESOURCE DESCRIPTION FRAMEWORK (RDF) With the standardisation of RDF in 1999 through a W3C Recommendation [5], RDF became a step towards realising a *machine-processable* Web. The RDF syntax is defined by the following terminologies:

Definition 3 *An **RDF terminology** $\mathcal{T}$ is the union of pairwise disjoint infinite sets of terms: [Klyo4].*

- *the set $\mathcal{I}$ of IRIs,*
- *the set $\mathcal{L}$ of literals (plain $\mathcal{L}_p$ or typed $\mathcal{L}_t$), and*
- *the set $\mathcal{B}$ of blank nodes.*

The set $\mathcal{V} = \mathcal{I} \cup \mathcal{L}$ represents the *names* used in RDF terms, and is called the *vocabulary*.

An RDF data model consists of *triples* or *statements* (see Table 2.1). An RDF triple t comprises three components, $t := (s; p; o)$, where, s is the *subject*, p the *predicate*, and o is the *object* of the triple. We describe each of these notions in the following:

- **Subject:** A resource over which an assertion is made is called a subject, and it can only be a IRI or a *blank node*. A blank node, (a.k.a. an *anonymous node*) is a unambiguous unique identifier in a local context and can be used as a IRI.
- **Predicate:** A binary relation through which two resources are connected. A predicate (a.k.a. property) can only be a valid IRI.
- **Object:** The value connected to the subject through the predicate. The representation of an object can be an IRI, a blank node, or a literal value (string, number, etc).

Therefore, a triple can be defined as:

Definition 4 ***RDF triple:** An RDF Triple $t := (s; p; o)$ is an element of the set $(I \cup B) \times I \times (I \cup L \cup B)$.*

Table 2.1: Representation of the author's details in triples or statements

prefix dsi:	<https://datascienceinstitute.ie/people/>
prefix foaf:	<http://xmlns.com/foaf/0.1/>

Subject	Predicate	Object
dsi:qaiser-mehmood	rdf:type	foaf:Person
dsi:qaiser-mehmood	foaf:name	"Qaiser Mehmood"@en
dsi:qaiser-mehmood	foaf:gender	"male"@en
dsi:qaiser-mehmood	foaf:birthday	"15-10-1984"^^xsd:date
dsi:qaiser-mehmood	foaf:skypeID	"qaiser400"

A finite set of triple is called an RDF graph. Thus, an RDF graph can be defined as:

3 Cool IRIs: https://www.w3.org/2004/11/uri-iri-pressrelease
4 RFC3987: https://tools.ietf.org/html/rfc3987
5 RDF: https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

Definition 5 ***RDF Graph:*** *An RDF graph* G *is a directed labelled graph* (V, E) *where the set of edges* E *is represented by a set of RDF triples. Nodes* V *can be IRIs* I*, blank-nodes* B *or literals* L*.*

An RDF graph can be serialized in different formats, such as *N-Triples*, *Turtle* or *RDF/XML*. The data in the following example is represented in the RDF Turtle format.

EXAMPLE 2.2. : Listing 4 shows an excerpt of RDF triples in Turtle format. This type of data format is easy to understand by humans, and *prefixes* are used to avoid the large representation of resources. In Listing 4, the prefix abbreviations foaf: and xsd: are used for http://xmlns.com/foaf/0.1/ and http://www.w3.org/2001/XMLSchema# respectively. Thus, any resource that is bounded to the namespace http://xmlns.com/foaf/0.1/ can be written as for example foaf:Person, foaf:name, etc.. The namespace http://www.w3.org/2001/XMLSchema# is used to represent the datatype of an object. For example the value of the property foaf:gender is of type xsd:string.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<https://datascienceinstitute.ie/people/qaiser-mehmood> a foaf:Person ;
foaf:name "Qaiser Mehmood"@en ;
foaf:gender "male"^^xsd:string ;
foaf:birthday "15-10-1984"^^xsd:date ;
foaf:skypeID "qaiser400" .
```

Listing 4: The Turtle representation of the author's details

2.2.3 Linked Data

The publication method of structured data on the Web is called Linked Data. Linked Data is one of the core pillars of the Semantic Web (a.k.a. the Web of Data). The publication of Linked Data is based on the following four principles which are known as the *"Linked Data principles"* [BHB11b].

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

Thus, Linked Data publishing is based on Web standards such as URIs, HTTP, and RDF discussed in previous sections. Linked Data is primarily used to share machine readable information and enables data to be connected and queried from different data sources.

In a note [6], Tim Berners-Lee mentioned that only putting data on the Web is not the Semantic Web, but it is about establishing links, so that human and machine agents can explore this data.

6 Note:- Linked Data: https://www.w3.org/DesignIssues/LinkedData.html

2.2.4 Linked Open Data (LOD)

Linked Open Data (LOD) [JCB11] is linked data which is publicly made available and that can be openly accessed. Tim Berners-Lee in 2010 suggested a five star deployment scheme [Ber5] for Linked Open Data that can be explained as:

"The rating begins at one star and the more proprietary formats are removed and links added, the more stars data gets" [Ont].

In the context of Semantic Web graph databases, the exploration of data from distributed data sources, connected through different links, enables the *inference* of new knowledge from existing facts. Therefore, more and more data from different organizations is appearing every day on the Web, as Linked Open Data, which is enhancing innovations in cognitive and semantic technologies. Figure 2.3 shows the Linked Open Data Cloud diagram [Lin], where datasets from different domains are represented. There are 1,255 datasets published and catalogued in the LOD cloud as of May, 2020.

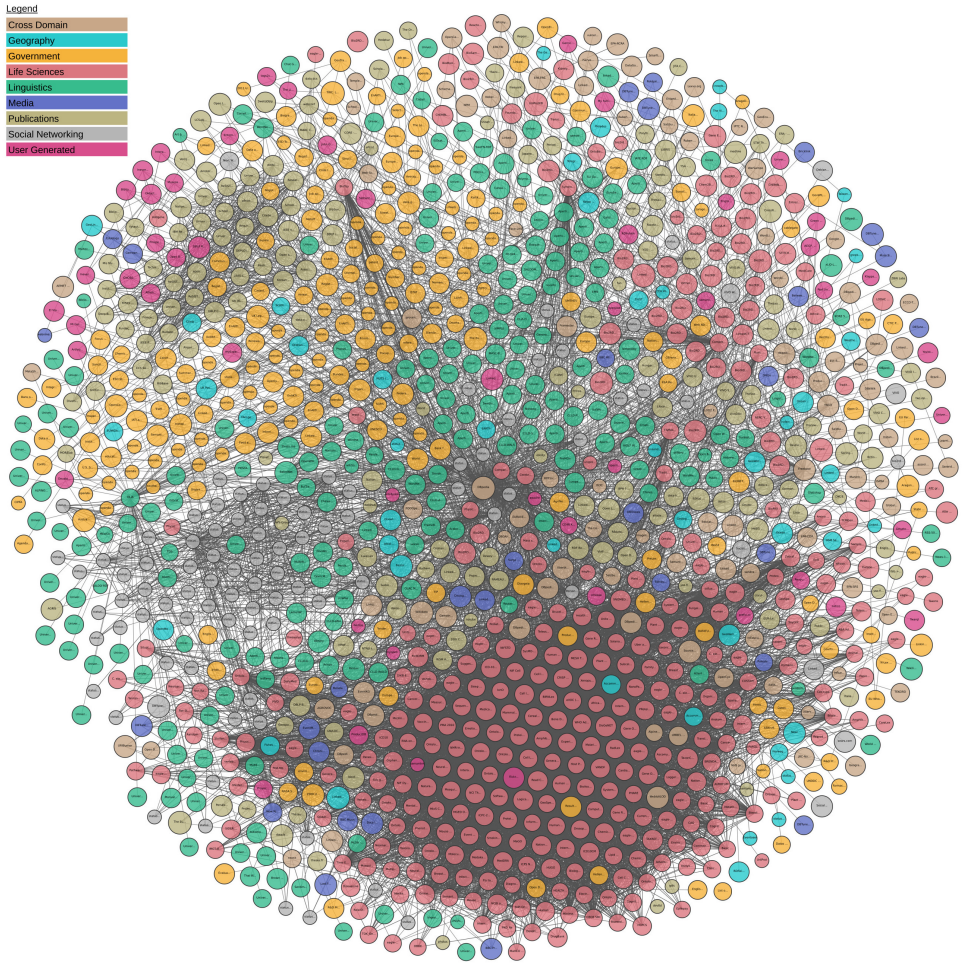


Figure 2.3: The Linked Open Data Cloud from lod-cloud.net [Lin].

The experiments conducted in this thesis are based on datasets publicly available on the Linked Open Data cloud.

2.2.5 RDF Storage and Indexing

Most of the data exposed as LOD is stored in different repositories. An RDF repository (a.k.a a Triplestore) is similar to a Relational Database Management System (RDBMS) which is capable of storing, indexing, and querying RDF data. The existing repositories, for storing RDF data, can be divided into two categories: *native* and *non-native*. The native triplestores are those that implement their own storage backend and are broadly classified as *main memory-based*, and *disk-based*. On the other side, triplestores that are using existing database systems are denoted as non-native. These storage systems are briefly described as follows:

- **In-memory storage:** is a viable solution for storing RDF triples depending on the availability of main memory. Fuseki server [Jena] provides the option to work as an in-memory storage system, and in this thesis, Fuseki is the storage system used as SPARQL endpoint for experiments.

- **disk-based storage:** is a solution to store RDF data more permanently on the file system. Systems such as RDF Triple eXpress RDF-3x [GN11] stores RDF data aggressively as compressed index. Other example for a customised datastore is Header Dictionary Triple (HDT), a self-indexed, compact and compressed RDF format. HDT is also used in this thesis as underlying database in chapter 4. We discuss about HDT in the following paragraph. Systems such as Stardog [Sir] and Blazegraph [Bla] also fall into disk-based storage. However, performance bottlenecks often appear with such systems.

  HDT [MAF12] is a compressed index representations of RDF graphs. In the following, a high level introduction is given of HDT, whereas a detailed description is available on the dedicated website[7].

  RDF HDT is a data-centric format which reduces verbosity in favor of machine-understandability and data management. HDT specifications provide a compact binary representational format that is optimized for storage, querying, and distribution over the network. An HDT-encoded dataset is composed of three logical components (Header, Dictionary, and Triples), carefully designed to address RDF peculiarities. Figure 6.1 depicts these three components of HDT representation.

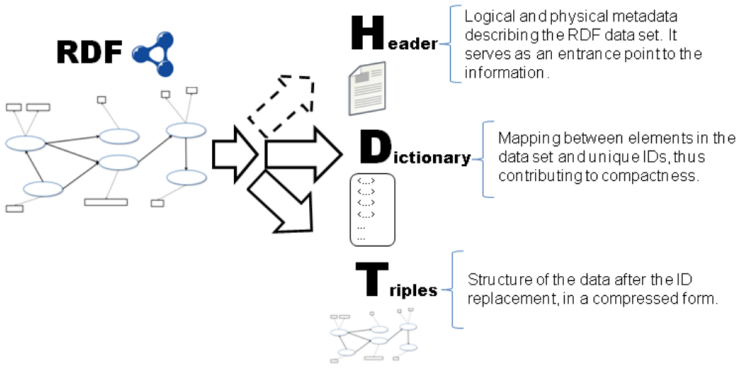


Figure 2.4: HDT components: Header-Dictionary-Triples [MAF12].

7 HDT: http://www.rdfhdt.org/hdt-internals/

- **Non-native storage:** makes uses of existing database management solutions to store data permanently. These systems benefit from years of research and development experience from RDBMS. For instance, Virtuoso [Teab] and Apache Jena TDB [Jenb] fall into this category.

Most of the triplestores which have the ability to store RDF data support the query language SPARQL, which is describe in Section 2.3.2. Those triplestores also use the SPARQL protocol [FWC+13] to provide SPARQL endpoints, which are query services for the SPARQL language.

2.3 QUERYING RDF GRAPHS

As discussed previously in Section 2.1.2, there exist different graph data models and to query these, different query models are used. For instance, as described above, to query data in RDF, SPARQL is used, which is based on pattern matching queries. Extensive work has been done on different languages, in terms of query features, query extensions, query performance, etc. This thesis is about navigational or path queries. Therefore, in this section we mainly focus on relevant terms. We present a comparison of different languages based on their supported features in Section 3.1 of chapter 3. This comparison motivates the need for the proposed solutions in this thesis.

The summary of this section can be categorise as follows:

1. Subsection 2.3.1 elaborates on the foundations for navigational queries.

2. Subsection 2.3.2 highlights an overview of some languages that have been or are being used to query RDF graph data. We then discuss the standardised SPARQL query language and its features. Particularly we discuss property paths, as this thesis is about querying paths.

2.3.1 Foundations of Path Query Languages

Declarative languages are used to query graph data, and one of the fundamental paradigm of those are path queries. This section, therefore, starts with the basic notions and concepts used in path queries. We explain each concept and notion based on an example depicted in Figure 2.5.

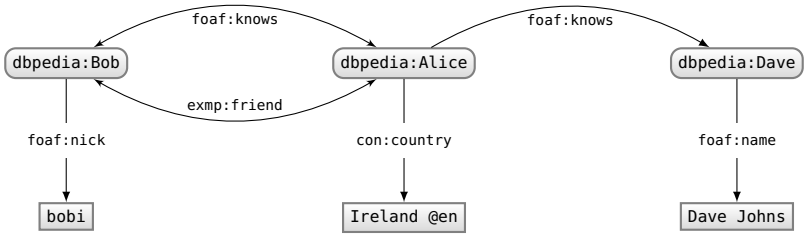


Figure 2.5: An RDF graph describing relations between people.

REGULAR PATH QUERIES (RPQS) RPQs queries [CMW87; CM90; CDL+03] evaluate paths of arbitrary lengths between pairs of nodes (i.e., s t) by means of the given *regular expressions* [Kle51]. The compositions of path patterns are based on the sequence of elements [MW95] permitted by regular expressions. The basic operation in RPQs are as follows [Prz17]: (i) parentheses, (ii) a Boolean logical operator (|) "or", and (iii) quantifiers (-,?, *, +, {n}, {min, max}). These constructs laid the fundamental basis for the navigational features in the SPARQL query language. A RPQ query on a graph G can be expressed as:

$$Q(s,t) = s \xrightarrow{L} t \text{ where:}$$

where

- L is a regular language or expression which is used over a fixed alphabet
- the result is the set of all pairs of nodes (u_i, u_j) in the graph G such that there exists at least one path P between s and t filtered by the L expression.

To present one of the RPQ operators (e.g., "+"), an example query where paths are filtered by a sequence of *knows* edges is described by

$$Q(s,t) = s \xrightarrow{knows^+} t$$

When the above query is evaluated on the RDF graph G shown in Figure 2.5, we retrieve the following results:

$$\{(Bob \rightsquigarrow Alice), (Alice \rightsquigarrow Dave), (Bob \rightsquigarrow Dave)\}$$

TWO-WAY REGULAR PATH QUERIES ((2)RPQS) Such queries [CM90; CDL+03] extend the RPQs by adding the *inverse* operator ("-") that allows "backwards" traversing of edges, as in the following example:

$$Q(s,t) = s \xrightarrow{(knows^+|knows^-)} t$$

Evaluating the above query on the graph G depicted in Figure 2.5, we can see the results filtered by "knows" relation in both directions:

$$\{(Bob \rightsquigarrow Alice), (Alice \rightsquigarrow Dave), (Bob \rightsquigarrow Dave),$$
$$(Alice \rightsquigarrow Bob), (Dave \rightsquigarrow Alice), (Dave \rightsquigarrow Bob)\}$$

CONJUNCTIVE REGULAR PATH QUERIES ((C)RPQS) Conjunctive RPQs [CM90] include the conjunctive queries (CQs) expression "∧". A query could be evaluating a path filtered for example by *knows* and *friend* edges. Such a query can be defined as:

$$Q(s,t) = (s \xrightarrow{knows^+} t) \wedge (s \xrightarrow{friend^+} t)$$

The above query, when evaluated on the graph G shown in Figure 2.5, will restrict the results to pairs of nodes connected through both "knows" and "friend" edges, therefore returning only one result:

$$\{(Bob \rightsquigarrow Alice)\}$$

Conjunctive Two-way Regular Path queries support both the "∧" operator and the "-" operator.

2.3.2 Query Languages for RDF data

The standard query language used to query RDF data is SPARQL [HSP13]. However, there is a variety of other query languages [OO02; KAC+02; KMA+03; Sir] that can also be used for that purpose. Some of them (e.g., [OO02]) use an XPath-like syntax, while others [KAC+02; KMA+03; BK03] are SQL-like declarative query languages. Before the acceptance of SPARQL as a W3C recommendation, there were two predecessors [MSR02; Sea04] of SPARQL used to query RDF graphs. In this thesis we focus only on SPARQL.

SPARQL is a W3C recommended [PS+17] declarative language used to query RDF – based on pattern matching – from SPARQL endpoints. The basic building block of a SPARQL query is a *triple pattern*. An example of SPARQL query is given in the introduction chapter as Example 1.1. A triple pattern is similar to an RDF triple (see Section 2.2.2), except that variables can be used at any position. A variable in SPARQL start with the character "?". Executing a SPARQL query, a mapping between the query triple pattern and the graph triples in the queried graph is created. Any variable in the query triple pattern acts as a wildcard and is matched against any node (if in subject or object position) or edge (if in predicate position) in the graph. A set of triple patterns is called *basic graph pattern (BGP)* where triple patterns are concatenated by "." (interpreted as a logical "and"). The result of a BGP is the intersection of the results of all triple patterns and their shared variables computed on graph triples. A BGP is more formally defined as follows:

Definition 6 ***Basic Graph Pattern (BGP):*** *A tuple* $t \in (I \cup L \cup B \cup V) x (I \cup V) x (I \cup L \cup B \cup V)$ *is a basic graph pattern, where I, L, B and V are disjoint sets of IRIs, literals, blank nodes and variables respectively. If P1 and P2 are basic graph patterns, then (P1 . P2) is a graph pattern [DuC11].*

In addition to BGP in SPARQL, the operators SERVICE, VALUES, and BINDINGS introduce the concept of *federation* where multiple sub-queries can be executed on remote endpoints. We briefly explain these operators in Section 2.4.2. A detailed explanation of SPARQL queries and their syntax can be found in W3C recommendation [PS+17].

More directly relevant to the work in this thesis, SPARQL1.1. [HSP13] recently introduced Property Paths (PPs) [Sea] to navigate in an RDF graph.

SPARQL PROPERTY PATHS (PPS) Standard SPARQL queries have limitation to navigate through the RDF graphs. The answer to a pattern matching query is the set of possible binding for all the variables. Therefore, a standard SPARQL query can not be constructed to find a path of arbitrary length. For this reason, a feature was introduced in SPARQL1.1 [HSP13] to make it possible to navigate between two nodes in a graph. Property Paths (PPs) are based on the notion of connectivity queries (see Section 2.1.2) and correspond to Conjunctive Regular Path Queries (C2RPQs) [CM90]. SPARQL 1.1 PPs are syntactic sugar, allowing for more concise expression of some SPARQL basic graph patterns, and adding the ability to match paths of arbitrary length. They are not necessary nor required and they do not change SPARQL queries, but they can make

queries shorter and easier to both write and read [Teac]. A trivial case to evaluate the path is a single triple pattern (i.e., *one hop path*). Property Path Pattern are defined as:

Definition 7 *A **Property Path Pattern*** $(s; PP; o)$ *is a path pattern where* $s \in V \cup I, o \in I \cup L \cup B \cup V$ *and* PP *is the property path expression described in Table 2.2.*

A property path expression is similar to a string regular expression. However, this applies only over properties. Unlike typical SPARQL queries, where predicates can also be variables, the property path expressions can not have variables at property/predicate location. Variables can only be used at the *start* or *end* of a path. PP queries determine all matches of a path expression and bind subjects or objects as appropriate [Sea]. Only one match is recorded - no duplicates for any given path expression, although if the path is used in a situation where its initial points is already repeated in a pattern, this duplication is preserved [Teaa]. PPs allow *cycles* in paths. Table 2.2 shows the standard expressions that are used in property path queries. In that table **uri** is a URI or a prefixed name and **elt** is a path element.

Table 2.2: Standard expressions used in property path queries.

Syntax Form	Matches
uri	A URI or a prefixed name. A path of length one.
^ elt	inverse path (object to subject)
!uri or !(uri1| ...|urin)	Negated property set. A URI which is not one of uri_i
!^ uri and !(uri1| ...|urij|^ urij+1| ...|^ urin)	Negated property set. A URI which is not one of uri_i, nor uri_{j+1}...^ uri_n as reverse paths)
(elt)	A group path elt, brackets control precedence.
elt1 / elt2	A sequence path of elt1, followed by elt2
elt1 | elt2	A alternative path of elt1, or elt2 (all possibilities are tried).
elt*	A path of zero or more occurrences of elt.
elt+	A path of one or more occurrences of elt.
elt?	A path of zero or one elt.
eltn,m	A path between n and m occurrences of elt.
eltn	Exactly n occurrences of elt.
eltn,	n or more occurrences of elt.
elt,n	Between 0 and n occurrences of elt.

2.4 DISTRIBUTED PROCESSING OF LINKED DATA

RDF graph data and the relevant management technique we have studied so far are characterized as "centralized approaches", which means that the storage and querying are performed over a single machine. However, with the proliferation of Linked Data, such architectures suffer from many limitations.

A *distributed database management system (DDBMS)* is a system where data management is distributed over a network of systems [CB14]. Figure 2.6 provides an overview of different types of DDBMS, organised hierarchically.

In homogeneous distribution strategy, all sites are implemented by similar systems. All the distributed nodes have the same software, configurations, and are aware of each others. A

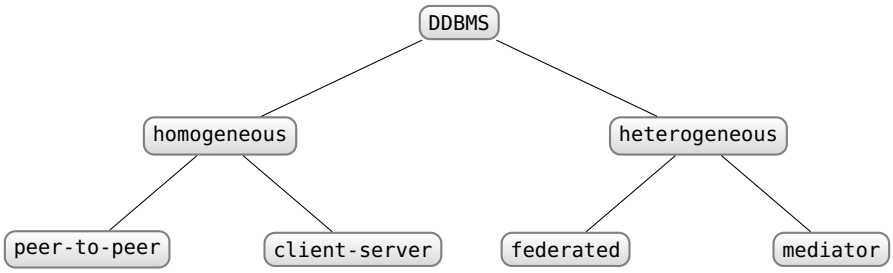


Figure 2.6: Taxonomy of the distributed Systems

homogeneous architecture appears to be a single system to the users. The architectural design is simple for such systems and follows the following conditions [Wik]:

- the structure of the underlying data distributed across the nodes must have the same model
- the database software deployed over each node or site must be identical

In contrast, the sites in heterogeneous use different types of schema. The nodes may not be aware of each other and also may have different data models, software, hardware, communication and access protocols [Wik]. In the context of RDF, heterogeneous distributed system can be distinguished into two main categories. One approach is mediator-based, which mediates between different systems. This approach corresponds to the *Ontology-based data access* (OBDA) model for example, which aims to exploit the semantic knowledge expressed in ontologies [CB14]. The other approach is called *federated*, and does not require any global schema.

2.4.1 Homogeneous systems

As explained before, homogeneous distributed systems use the same schema, software, configurations in all nodes. Therefore, the system's architecture can be viewed as a centralized database where different sites or nodes transparently handle the user requests and processing tasks. That is why this architecture is easy to implement and manage as compared to the heterogeneous systems. The query processing in homogeneous architecture can be increased by adopting *parallel* processing, which is difficult in the case of heterogeneous systems where different software may be installed on various nodes. In homogeneous systems, the consideration should only be focused on data fragmentation and distribution, because the performance of this architecture majorly depends on these two factors.

PEER-TO-PEER (P2P) In P2P systems, there does not exist any centralized control and each system in this architecture is built and treated equivalently to others. Every node can act as *server* or *client* at the same time and is able to communicate with other nodes. The drawbacks of this architecture is the performance issue for query requests, because the queries must be flooded in the network for a targeted resources [CB14]. In the context of RDF data management in P2P network, both triples and indexes (triplestores URIs) are distributed and maintained over each node. The first P2P based RDF stores and architectures, for instance [NWS+04], were designed using unstructured P2P networks. However, such systems do not have any source selection mechanism because of the lack of index. Therefore, to retrieve query answers, the request is dispatched to each system in the network. We present and explain an approach in Chapter 5,

where we implemented a kind of P2P-based distributed pathfinding architecture, which solves the above-mentioned issue.

CLIENT-SERVER In client-server (a.k.a master-slave) architecture, the client sends the request to the server. In response to each request, the server returns the results to that particular client. There could be many possibilities, in terms of variants, to implement this architecture. For example, in a very strict client-server architecture, there must be only one server and clients have fixed roles. Clients can interact with each others, however, queries are only answerable by the server. In RDF data management, several query processing approaches are using the MapReduce [DG10] paradigm. For instance, Trinity [SWL13], Giraph [Gir], Hadoop [SKR+10], Spark [ZCF+10] are example of such distributed systems that follows this master-slave architecture. A detailed understanding of client-sever systems can be obtained from the book [CB14].

2.4.2 Heterogeneous systems

As noted before, unlike homogeneous systems, the heterogeneous architecture does not restrict nodes into having the same schema and database software.

QUERY FEDERATION The concept *query federation* for RDF data is motivated by the fact that LOD data is mostly owned and controlled by independent data providers. Query federation allows to query the distributed data stored on heterogeneous systems. To query such types of data, different query federation engines have been proposed. Those include Fedx [SHH+11], HiBISCus [SN14a] and BioFed [HMZ+17], which can query data from different SPARQL endpoints. Query federation systems can scale easily and data management is simpler. The query federation engine can refer to new data sources in the submitted queries without requiring centralised management of the data. It only retrieves intermediary results from nodes and combine them.

With the advantages of query federation engines, there are also challenges that must be considered when designing a federated engine. The relevant data source need to be identified for a given query to answer it efficiently. On the other side, in homogeneous systems, data is partitioned and distributed in a controlled way. However, in chapters 5 and 7 we argue that data source selection in homogeneous and controlled environment can also be adopted, and can be useful for efficient query processing. In federated systems, there have been extensive work done, for instance [SN14b; QL08a], related to data source selection and query performance. Finally, a federated system should try to reduce the number of requests to data sources.

There are several type of federated engines [LKM+11; AVL+11; SHH+11; GS11a; SN14a; MSM+15; HMZ+17] and these can be classified as:

- **indexed-based** systems maintain indexes either for source selection or query optimization
- **index-free** systems do source selection on-the-fly by probing the data sources.

In this thesis, Chapter 6 presents an indexed-based approach to federated PP queries, and Chapter 7 briefly explains an index-free approach to find paths in distributed environments.

SPARQL AND QUERY FEDERATION A query federation engine in nutshell is a unique interface for interacting and querying multiple data sources. Most of the RDF databases allow their data

to be accessed through the SPARQL protocol, and such systems are called SPARQL endpoints. SPARQL 1.1. introduced the keyword SERVICE that allows us to send subqueries to remote endpoints. Query federation is basically a virtual integration of RDF datasets. The concept of query federation overcomes and tackles the issue that often are faced in centralized approaches. Some of the challenges in centralized approaches are for example; (i) querying multiple graphs requires first to merge them into a single graph, and this is a cumbersome task; (ii) locally retrieved data that is being updated at the source need to be synchronised; and therefore (iii) the merged data might not be as up-to-date as the original data source. Thus, with the proliferation of Linked Open Data, it has become an essential requirement to develop scalable, distributed and federated querying techniques to efficiently query RDF data over multiple distributed graphs. A SPARQL federated query in Figure 5 shows an example where, using SERVICE and PP syntax, we can find the population of different countries from a remote endpoint.

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?country ?pop WHERE {
VALUES ?country {
                dbr:Ireland
                dbr:France
                dbr:England }
-- call to remote endpoint
SERVICE <http://dbpedia.org/sparql> {
-- a property path syntax (| or alternate) on predicate position
?country dbp:populationCensus | dbo:populationTotal ?pop .
}
}
```

Listing 5: SPARQL federated query

In the context of graph traversal or navigation, we have explained in Section 1.1 that standard SPARQL queries cannot navigate paths of arbitrary lengths. Therefore, the SPARQL property paths feature was introduced. However, there was still a research gap in the context of SPARQL path query federation: The SERVICE clause is not feasible in path federation due to, for example, slow performance, query complexity, and result completeness issues. This thesis gives an overview of a federated technique (see Chapter 6) for querying federated paths over distributed Linked Data sources.

MEDIATOR-BASED Mediator-based approaches are also like federated systems, which provide the single interface to query different data sources. Unlike normal query federation, where federated databases have identical data models or underlying schema, the mediator systems deal with different data models. In such type of systems, an integrated schema that integrates with different schema of various data sources has to be provided. These systems include the ones relying on the ODBA approach. The targeted data sources generally correspond to RDBMSs. However, recently some research have been initiated towards using XML, NoSQL, CSV stores [KZJ+19; AKC+19]. The integrated schema is usually represented as an ontology (e.g., OWL2QL, R2ML, and etc.) The mediator-based approaches make sense in ODBA systems where data is produced

in different formats. The importance of mediator-based approaches have recently been highlighted by database experts where they predicted the demise of *"One Size Fits All"* approaches used in the data retrieval and management solutions [SÇ18].

3 STATE OF THE ART AND RELATED WORK

A brief background on querying Linked Data, both for centralized and distributed management, is presented in the previous Chapter 2. Hereon, we will discuss the state of the art and related work. This includes data distribution and management, as well as centralized and distributed graph and query processing. In the context of querying Linked Data, many languages have been introduced and some of them gained traction, such as SPARQL. A fair amount of work has also been done to extend the already existing standards. In this chapter, we discuss SPARQL extensions (particularly SPARQL property paths). We start this chapter with a feature-based comparison of different languages and extensions. We then briefly discuss the state of the art both in centralized and distributed architectures. We highlight the benefits and limitations of these approaches and compare them with our proposed solutions.

3.1 RDF QUERY LANGUAGES AND FEATURED-BASED COMPARISON

Table 3.1 shows a feature-based comparison of different query languages and some extensions of existing languages used to query semi-structured data. Table 3.1, is inspired and adapted from [Prz17]. The authors in [Prz17] adopted seven features of three query languages from the work in [AGH04; AG05] and nine from [HBE+04; VCM08]. We move the abstract level explanation of each feature and their *source of origin* to Appendix A.

The additional features and languages we added in Table 3.1 are: (i) *K-Shortest Paths*, and (ii) (i.e., RDF3X, gremlin, GraphQL, RDF3X-paths, Stardog-Paths, and our SPARQL path query extension [SMU+17]) respectively.

From Table 3.1 it can be concluded that all query languages focus on a subset of features, and therefore cannot fulfil all possible combinations of requirements: there does not exist a *one-size-fits-for-all* language. As this thesis covers path queries that are a fundamental paradigm in graph traversal, we are mostly interested in SPARQL2L, SPARQLeR, RDFPath, Stardog-Paths, RDF3X_Paths, and Top-k-Path, as the only query languages that evaluate not only paths but also output and enumerate complete paths. Some of the languages highlighted are extensions of others; for instance, nSPARQL, SPARQL2L, SPARQLeR, Top-k-Path are direct extensions of the SPARQL query language.

A few of the query languages shown in Table 3.1, have also been evaluated in distributed query processing. For this, some of those languages have introduced either specific features (e.g., SERVICE in the context of SPARQL) or were used as a query template for query engines or the underlying algorithms. Examples are RDFPATH, TriAL, and our proposed extension [SMU+17].

Table 3.1: A comparison of querying features of languages for RDF, where "×" indicates no support, "○" partial support, and "●" full support. The table is inspired by numerous previous studies such as [Prz17; Abi97; HBE+04; AG05; AG08; VCM08]

Language	Basic					Navigational												Analytical				Relational				Others	
	Namespace	Constraints	Language	Lexical Space	Value Space	Graph Pattern	Adjacent Resources	Adjacent Predicates	Fixed-length Path	Path Variable	Inverse Path	Non-simple Path	Recursion (Regular Expression)	Constrained Regular Expression	Optional Pattern	Entailment (Reasoning)	Querying Topology	Output Paths	Degree of Resource	Distance between Resources	Shortest Path	K Shortest Paths	Union	Difference	Aggregation	Sorting	Closure	Collections & Containers
RQL	●	●	×	●	●	●	○	○	●	×	×	×	○	×	○	●	×	×	○	×	×	×	●	●	●	×	×	●
SeRQL	●	●	●	●	●	●	○	×	○	×	×	×	×	×	●	●	×	×	×	×	×	×	●	●	×	×	●	○
RDQL	○	●	×	●	○	●	○	○	○	×	×	×	×	×	×	○	×	×	×	×	×	×	×	×	×	×	×	○
Triple	●	●	×	●	×	●	○	○	○	×	×	×	●	×	×	○	×	×	×	×	×	×	●	×	×	×	×	×
N3	●	●	●	●	×	●	○	○	○	×	×	×	●	×	×	○	×	×	×	×	×	×	●	×	●	×	●	●
Versa	×	●	×	●	×	●	○	○	×	×	×	×	○	×	●	×	×	○	×	×	×	×	●	○	●	●	×	○
Corese	●	●	●	●	●	●	●	●	●	●	×	×	×	×	●	●	×	×	×	×	×	×	●	●	○	×	×	○
SPARQL2L	●	●	●	●	●	●	●	●	○	●	×	●	●	○	●	×	×	●	×	×	×	×	●	●	×	×	○	○
SPARQLeR	●	●	●	●	●	●	●	●	○	●	●	×	●	○	●	○	×	●	×	×	×	×	●	●	×	×	○	○
LOREL	×	●	×	×	●	●	●	○	●	●	×	×	●	×	×	×	×	○	×	×	×	×	●	×	●	×	×	×
G+	×	×	×	×	○	●	●	●	●	×	●	×	●	×	×	×	×	●	●	●	×	×	×	×	×	×	×	×
GraphLog	×	×	×	×	○	●	●	●	●	×	●	×	●	×	×	×	×	○	●	●	×	×	×	×	×	×	×	×
StruQL	×	×	×	×	○	●	●	●	●	×	×	×	●	×	×	×	×	○	●	●	×	×	×	×	×	×	×	×
SP.1.0	●	●	●	●	●	●	●	●	●	×	×	×	×	×	●	×	×	×	×	×	×	×	●	●	×	●	○	○
SP.1.1	●	●	●	●	●	●	●	●	●	×	●	×	●	×	●	○	○	×	●	×	×	×	●	●	●	●	○	○
RPL	●	●	●	●	●	●	●	●	●	×	●	●	●	○	●	●	×	○	×	×	×	×	●	●	×	×	×	○
PSPARQL	●	●	●	●	●	●	●	●	●	●	×	●	●	×	●	●	○	×	×	×	×	×	●	●	○	×	○	○
cpSPARQL	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	×	×	×	×	●	●	○	×	○	○
nSparql	●	●	●	●	●	●	●	●	●	×	●	●	●	●	●	●	○	×	×	×	×	×	●	●	○	●	○	○
TriQ	×	●	●	×	×	●	●	●	●	×	●	●	●	●	●	●	●	×	×	×	×	×	●	●	×	×	×	○
TriAL	×	●	●	×	×	●	●	●	●	●	●	●	●	●	●	●	●	×	×	×	×	×	●	●	×	×	●	○
RecSPARQL	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	×	●	×	×	×	●	●	●	●	○	○
RDFPath	●	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	○	●	●	●	○	×	●	●	●	●	○	○
TriAL-QL	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	×	×	×	×	●	●	×	×	●	○
Stardog-Paths	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	×	×	●	×	●	●	×	×	●	○
Gremlin	●	●	●	●	●	●	●	●	●	×	●	●	×	×	●	●	○	×	●	×	●	×	●	●	●	●	●	●
RDF3X_Paths	●	●	●	●	●	●	●	●	×	●	×	●	×	○	○	×	×	●	×	×	●	×	●	○	●	●	○	○
ESWC-Top-k-Path	●	●	●	●	●	●	●	●	●	●	×	×	○	×	●	×	×	●	○	●	●	●	●	●	●	●	○	○
Top-k-Path	●	●	●	●	●	●	●	●	○	●	●	●	●	○	●	×	×	●	●	○	●	●	●	●	●	●	○	○

It is worth mentioning that we used our proposed extension [SMU+17] as query template for distributed query processing in chapters 5, 6, and 7.

3.2 CENTRALIZED APPROACHES:

As shown in Table 3.1, we can see that different query languages that are proposed to query Linked Data differ in the features they offer. For instance, out of these 28 languages, only (i) Stardog-Paths, (ii) Top-k-Path [SMU+17], (iii) SPARQL2L, (iv) SPARQLeR, (v) RDF3X_Path, and (vi) RDFPath evaluate not only the paths but also output and enumerate the complete paths. On the other hand, G+ is also a query mechanism that has the ability to calculate and output the paths. However, G+ is not a declarative query language. For the purpose of brevity, we focus only on those which are more relevant to our work. In the following section we highlight SPARQL extensions and their limitations.

SPARQL EXTENSIONS As mentioned in Chapter 1, path search is among the most studied problem – with many real-world applications – in computer science and is also central in the Semantic Web and Linked Data area. Although SPARQL has emerged as the standard query language for RDF data, in its first specification (SPARQL 1.0) it provided limited support for navigational or path querying. Users were only able to find up to a certain length of path by mean of conjunctive triple-patterns. This limitation motivated several SPARQL extensions to address and propose more expressive navigational queries with arbitrary lengths.

- **SPARQLeR [KJ07] and SPARQL2L [AMS07]**

  These are the first proposals for SPARQL path query extensions, and both of these introduced a so-called *path variable* (prepended by % or ?? instead of the standard single ?). For example, SELECT %path $\{< r > \text{\%path} < s >\}$ is a graph pattern to match any path between r and s. Additionally, a user can filter paths based on their variables and retrieve complete paths. This was a valuable functionality introduced in SPARQL to find paths of arbitrary lengths. However, the drawback in both languages is the inconvenient and inconsistent syntax. To describe a desired path pattern, the *filter function* or so-called *meta-properties* are the only options. Regular expressions can only be applied on path variables (e.g., regex(%path, $*$) using the filter function (i.e., FILTER regex(%path, $*$)). Both languages were introduced with a focus on their prototype implementations. Moreover, they support only simple path querying. Nevertheless, both SPARQLeR and SPARQL2L were the first two languages which mitigated the shortcomings of SPARQL 1.0 and highlighted the concept of querying and outputting paths with arbitrary lengths.

- **nSPARQL [PAG10], PSPARQL [CEG+11] and its extension cpSPARQL [ABE08]**

  These are more recent extensions of the SPARQL query language for path queries. The query language nSPARQL introduced the notion of *nested regular expressions* which combine regular expressions with an inverse operator, the notion of branching borrowed from XPath [CD+99], and nested existential tests. Having these, nSPARQL can be used to query structural properties along with other interesting queries that can not be captured by paths alone. One example of such queries could find, for instance, the chain of friend relationships with the constraint that friends should live in the same country.

  The main difference with *conjunctive regular path queries* ((C)RPQs) (see Section 2.3.1 of Chapter 2) is that paths retrieved can be of arbitrary lengths, whereas the number of conjunctions in ((C)RPQs) are always fixed. To illustrate that, consider how to simulate the

friends chain query in ((C)RPQs). This can only be evaluated by conjunctions of successively extended paths, where, in our example, a variable is required to check the existence of information about a country against each user. One of the core drawback of nSPARQL is that it can only assess the existence of paths, and not retrieve those paths as offered by SPARQLeR and SPARQL2L. However, the syntax and semantics of nSPARQL are well defined and a detailed evaluation is provided.

PSAPRQL is a language that provides meaningful results and a well defined query semantics. Like standard SPARQL, PSPARQL allows variables at predicate positions in path queries. In this way, users can retrieve *individual* intermediate elements of the path rather than retrieving only the first and last resource. The authors extended their PSPARQL version to cpSPARQL [ABE08] by adding constraints on regular expressions.

- **RDF3X_path [GN11] and Startdog [Sir]**

  A SPARQL path extension has also been reported in the RDF3X engine [NW10]. The syntax of RDF3X_path is similar to ones of SPARQLeR and SPARQL2L, and is only able to find a single shortest path. Users cannot find paths of arbitrary lengths. One of the most restrictive aspects of using RDF3X_path queries is that it is tightly incorporated into the RDF3X code-base and cannot be used independently: There is no possibility to query the raw RDF data, and data must be in RDF3X format. Further there is no description available, in terms of usage, results, performance, etc.

  More recently, Stardog [Sir] has been extended with a component named Pathfinder, with which a user is able to easily express a pathfinding query in SPARQL. Unlike SPARQL property paths, using Stardog a user can retrieve all nodes involved in a path between a source and a target node. However, they also did not provide details about the syntax and the semantics of this extension, and mentioned they would provide it later [Sir].

- **More expressive, easy-to-adopt and plugin-based querying solutions**

  In recent works related to graph analysis, it has been recognised that their exist further properties which are important for analytical purposes, that are not captured in previously mentioned works. One of the important features is finding top shortest paths, providing a number K of expected results. However, in the context of the Semantic Web this has not gain much attention so far. In fact to the best of our knowledge the three ESWC'16 [1] Challenges "Top-K Shortest Path in Large Typed RDF Graphs Challenge" submissions [HSJ+16a; HQI+16; DVM16a] were the first to deal with the top-K path computation task specifically. In [DVM16a] the authors focused on decentralised computation of top-K shortest paths using Triple Patterns Fragments (TPF), which we also compare with one of our approaches in Chapter 7. The authors in [HQI+16] developed an extension using an algebraic algorithm based on matrix multiplication and a special index structure. However, their solution did not support SPARQL property paths. Recently, a work published by Julien et al. [ASM20] explains how to execute SPARQL property path queries online and get complete results. Their idea is to break property path queries into Basic Graph Patterns (BGPs) sub-queries. However, their approach does not support path enumeration, neither top-K paths.

1 ESWC'16: http://2016.eswc-conferences.org/top-k-shortest-path-large-typed-rdf-graphs-challenge.html

On the other hand, the winner of the ESWC'16 Challenge Herlting et al. [HSJ+16a] not only extended the Eppstien routing algorithm but also extended the SPARQL property paths semantics while keeping the SPARQL query language syntax and expressiveness. They also embedded their implementation into a SPARQL endpoint. Their idea to extend property paths was to create the desired number of hidden variables containing resources and properties alternatively. They are numbered consecutively starting from zero. The length of a path is assigned to the ?length variable. However, their approach to extend property paths did not comply with the semantics of SPARQL property paths in the sense that we can not get paths of arbitrary lengths. This is because, as they mentioned, the hidden variable is used in their code implementation and is fixed. Moreover, they do not implement a complete list of standard regular expressions (as shown in Table 2.2 of Chapter 2). We also present a comparison, in terms of the trade-off between memory and performance, with [HSJ+16a] in Chapter 7 and show the benefits of distributed computation where local computation becomes a bottleneck.

3.2.1 Summary

We discussed above different approaches and query languages used for querying paths in a centralized way. We highlighted that all query languages focus on a subset of features and therefore can not fulfill all possible combinations of requirements for graph analysis. For instance, although SPARQLeR [KJ07] and SPARQL2L [AMS07] output and enumerate paths, they do not follow the SPARQL query semantics. On the other hand, nSPARQL [PAG10], PSPARQL [CEG+11] and its extension cpSPARQL [ABE08] follow the SPARQL semantics but cannot output the paths. They can only check the existence of paths. With RDF3X_path [GN11], we can find a single shortest path. However, this is tightly incorporated into the RDF3X code-base and cannot be used independently for RDF raw data.

Recently SPARQL1.1 introduced property paths (see Section 2.3.2 in Chapter 2). Although it provides the functionality to check the existence of paths of arbitrarily lengths, it cannot enumerate paths.

Considering the work done previously and keeping in mind an important property for graph analysis that was not captured in previously mentioned works, we propose an extension, Top-K-Path [SMU+17], in Chapter 4 to SPARQL1.1 property paths. Top-K-Path is fully compliant with the syntax and semantics of SPARQL and is more expressive than others approaches discussed previously. Our implementation extending the Jena ARQ [Teaa] framework is a basic, but robust solution for the K shortest path problem since we implemented bidirectional search. Our implementation works both for compressed data (Header Dictionary Triples (HDT), a compact data structure and binary serialization format for RDF) but also for raw RDF. In contrast to Herlting et al. [HSJ+16a], our solution can find K paths of arbitrary lengths.

3.3 DISTRIBUTED APPROACHES

In recent years, with the increasing availability of large graph datasets, executing complex queries or traversing large scale graphs has shown to be impractical. This is particularly true

when a single system is used with limited memory, creating a bottleneck in the processing of large volumes of data. Therefore, to process such data, there is a need to distribute it and execute queries in a distributed manner with a high degree of parallelism. In contrast with the centralised systems, discussed in the previous section (3.2), distributed architectures are characterized as higher processing capacity systems with large aggregated memory size. There are several distributed architectures and approaches, of which we discuss the most relevant ones to our work.

3.3.1 Hadoop-based approaches

To process big data, hadoop and its echo-system have become the *de-facto* standard both in academia and industry. In the context of RDF graphs, there have been numerous works (e.g., [HMM+11; KKT+12; PKT+12; PTK+14]) done on RDF data management using this platform. Surveys such as [KM15; YC10a] talk about such approaches in details. Many of them, for example [PSH+11; BWY17] in path querying, follow the MapReduce paradigm and use HDFS [RS10; HMM+11] as their underlying data structure to store RDF triples. In the MapReduce paradigm, when a query is issued against such systems, HDFS files are explored and scanned to find adjacent nodes, which are then concatenated using the MapReduce join implementation (for more details see [LOÖ+14]). Approaches which follow the MapReduce paradigm differ mostly with respect to how the RDF data is stored and accessed from HDFS files, and how many MapReduce jobs are used.

THE MAPREDUCE PROGRAMMING MODEL The MapReduce programming model was introduced by Google [DG10]. This model is scalable, and fault-tolerant and new nodes can easily be added to the cluster. Furthermore, the data distribution and parallel processing of the jobs is handled automatically. Developers do not need to take care of all these steps, therefore, they do not face the classical problems of data distribution (e.g., synchronization, network protocol, and etc.) The advantages and benefits of this programming model have led to a large number of applications built using it.

However, for most of the users this programming model is too low level: It does not provide a declarative syntax to execute the tasks, and therefore, even a simple computation has to be configured and programmed as Map and Reduce steps. To overcome these limitations, there have been a number of works done [ORS+08; MYL10; HKK+10; SPH+11] to provide declarative query mechanisms on top of the MapReduce process.

In the context of path queries, Martin et al [PSH+11] proposed an approach named RDFPath; a path query processing method on large RDF graphs with MapReduce. Their approach is inspired by XPath [CD+99] and designed according to the MapReduce model. A query in RDFPath is composed of a sequence of *location steps* where the output of the i^{th} location step is used as input of the $(i+1)^{th}$ location step [PSH+11]. Conceptually, more edges and nodes are added by each location step to the intermediate path and this path can be restricted by applying filters. However, in this approach, the distance between two nodes and the shortest paths are only partially supported. Further, RDFPath only supports paths of a fixed maximum length.

CHALLENGES AND LIMITATIONS IN HADOOP-BASED APPROACHES

i) Most of the systems are homogeneous distributed systems. That is, they work like a controlled environment where every distributed system or database must have the same software and configurations. Furthermore, the partition and distribution of data must follow certain procedures or constraints.

ii) Although some work has been proposed to execute SPARQL queries against such systems, most of the proposed approaches require procedural programming to construct a query because they do not support declarative queries.

In Chapter 7 we propose a solution that addresses these limitations and provides an optimized way to query the paths in distributed environment in an efficient way.

3.3.2 NoSQL-based approaches

Besides the hadoop ecosystem, there has also been work done related to RDF graph query processing in HBASE [2]. HBASE is a NoSQL distributed data store. The work in [KKT+12] provides a criterion to store and query RDF triples in HBASE. It is based on some permutations of subjects, predicates and objects which build indices that are then stored in HBase. Trinity [SWL13] is an other example of NoSQL-based systems. Trinity is built on a distributed memory-cloud, and model RDF data in its native format (i.e., entities as node and relationships as edges). It allows SPARQL queries to be executed in an optimized way, and also support graph analytics (e.g., reachability) on RDF data. Survey papers [SM20; CEF+13] highlight some prominent NoSQL databases, their characteristics and usage for storing and querying RDF data.

Trinity [ZYW+13] is also claimed to support random walks, regular expression, and reachability queries. Distributed NoSQL graph databases such as Neo4J [Neo] and Stardog [Sir] are optimized for graph navigation features or path querying.

CHALLENGES AND LIMITATIONS IN NOSQL BASED DISTRIBUTED DATABASES

Although some of the above mentioned NoSQL databases support querying distributed data, they still miss some important features, highlighted in Table 3.1, and have no capabilities for navigational queries across multiple graphs.

3.3.3 Partition-based approaches

The partition-based approaches [GHS14; GSM+14; PB17; LLT+13; HAR11; LL13] partition a large RDF graph into several fragments and distribute those fragments at different sites. Each remote site hosts its own centralized RDF storage and querying mechanism. To answer a posed query, it is first decomposed into multiple subqueries which are distributed across the sites. Each site answers the received query locally, and results from different sites are then aggregated.

In [LLT+13], an RDF graph is partitioned into n fragments, and each fragment is extended by including the N hop neighbors of boundary vertices. Their partitioning strategy restricts query processing in such a way that each decomposed subquery cannot be larger than N.

2 HBASE: http://hbase.apache.org

Partout [GHS14] is an engine to query graph data which uses its own partitioned strategy. It extends the concept of minterm predicates and uses the results of those predicates as its fragment units.

Lee et. al [HAR11; LL13] in their approaches proposed a "vertex block" concept where they define a partition unit as a vertex and its neighbors. They adopted a set of heuristic rules for distribution of those vertex blocks. A query is transformed into blocks and these blocks can be executed among all sites in parallel and without any communication.

TriAD [GSM+14] adopted METIS [KK95], a multilevel graph partitioning technique to divide the RDF graph into many partitions. Each partition is considered as a unit and distributed among different sites. At each site, TriAD maintains six large, in-memory vectors of triples, which correspond to all subject-predicate-object permutations of triples. It also constructs a summary graph to maintain the partitioning information.

CHALLENGES AND LIMITATIONS IN PARTITIONED-BASED APPROACHES Like hadoop, all of the above methods require partitioning and distributing the RDF data according to the specific requirements of their approaches. However, in some applications, the RDF repository partitioning strategy is not controlled by the distributed RDF system itself. There may be some administrative requirements that influence the data partitioning. For example, in some applications, the RDF knowledge bases are partitioned according to topics (i.e., different domains) or are partitioned according to different data contributors.

3.3.4 Federated SPARQL query systems

SPARQL queries when run over multiple distributed heterogeneous SPARQL endpoints are considered federated queries. In the context of Linked Data, different repositories may be interconnected, and therefore provide a virtual integration of multiple data sources. A common technique in query federation is the precomputation of metadata for each SPARQL endpoint. Based on the metatdata, the original query is decomposed into subqueries in a way that each subquery is dispatched only to its relevant endpoint. On receiving answer from each subquery, the results are aggregated or joined together to answer the original query.

In [QL08b], the metadata correspond to a service description that tells which triple patterns (i.e., predicate) can be answered. SPLENDID [GS11b] is a federated system that uses Vocabulary of Interlinked Datasets (VOID) as the metadata schema for describing endpoints. HiBISCuS [SN14c] relies on capabilities to compute the metadata. For each source, HiBISCuS defines a set of capabilities which map the properties to their subject and object authorities.

The systems above are categorised as *index-based*, i.e. they rely on the availability of an index of metadata about the federated endpoints. In contrast to these, FedX [SHH+11] does not require any index or preprocessed catalogue, but it sends ASK requests to collect the metadata on the fly. Based on the results of ASK requests, subqueries of the original query are dispatched towards the relevant endpoints.

Saleem et al. [SKH+16] provide an extensive, fine-grains evaluation of existing query federated engines. They show the impact of different factors such as: number of sources selected, number of ASK requests, etc. They conclude that source selection time significantly affect the overall query performance. In chapters 5, 6 and 7 we have introduced source selection mechanisms for each of our approaches to distributed path computation.

CHALLENGES AND LIMITATIONS IN FEDERATED-BASED APPROACHES With the proliferation of Linked Open Data, the size of the LOD cloud (see Section 2.3 of Chapter 2) is constantly increasing to reach hundreds of available datasets. Thus, finding a relevant data source for a federated query is a challenging task. For "normal" federated SPARQL queries, a lot of work has been done to find efficient methods for source selection. However, in the context of graph analytics, none of the existing approaches address path queries and the problem of source selection for path queries.

3.3.5 Vertex-centric approaches

There have been many distributed graph processing systems [Kos00] developed to efficiently analyse large graphs. We focus only o those [Gir; MAB+10; YC10b] which support navigational features. These systems are built on top of a cluster of low-cost commodity PCs and are deployed with a shared-nothing distributed computing paradigm.

Google's Pregel [MAB+10] is an example which adopts a vertex-centic computing paradigm. As discussed in Section 3.3.1, MapReduce is one of the popular approaches used in distributed graph processing. However, in-contrast with MapReduce, the Pregel-based approaches are inspired by *bulk synchronous parallel* models [Val90] and are more fault tolerant, scalable, and efficient.

Different approaches (e.g., [NS16; XWJ+18; WWX+19]) have adopted Pregel's architecture for distributed path or reachability queries. Nolé et al. [NS16] for example, proposed an evaluation technique according to the Pregel-based vertex-centric model. In their approach, at each step of the computation, each vertex derives an input query according to the symbol labeling of outgoing edges and propagate derivative queries to its neighbours.

In [XWJ+18; WWX+19], the authors proposed a distributed Pregel-based approaches named *DP2RPQ* to answer provenance-aware *regular path queries* RPQs (see Section 2.3.1 of Chapter 2) using *Glushkov automata*. In their approach, at each superstep, one hop of edges in the path of the RDF graph is matched forward to obtain intermediate partial answers. Also they designed different optimization strategies e.g., reduce the vertex computation cost and edge-filtering, to improve the overall performance of computation.

CHALLENGES AND LIMITATIONS IN VERTEX-CENTRIC APPROACHES As explained above, Pregel's vertex-centric computing model has been widely adopted in most of the recent distributed graph computing systems [Gir; SW13; NS16; XWJ+18; WWX+19] because of being fault tolerant and also efficient as compared to Hadoop-based approaches. In Pregel, message passing is done through vertex-to-vertex channels and therefore often causes bottlenecks in communication when processing real-world dense graphs (particularly the Linked Data where graphs mostly are multigraphs and nodes have high degrees).

3.3.6 Link traversal approaches

In graph processing paradigms, systems based on Hadoop, Partitions, and Vertex-centric computation are considered to be *homogeneous* infrastructures or models (see Section 2.4.1 of Chapter 2). Such systems work in controlled environments where each node of the distributed system can

have the same configuration and the distribution of the data can follow the specific requirements of the system. However, Linked Data by nature is based on heterogeneous distributed systems, exposed by different data providers using different systems, each holding data based on requirements external to the distributed query processing approach. Consequently, a focus on a paradigm called Linked Traversal has emerged in this context, which makes use of the characteristics of Linked Data. In order to execute a given Linked Data query, for example a path query, live exploration systems perform a recursive URI lookup process during which they incrementally discover further URIs that also qualify for lookup [UHP+15; Har16]. Such live explorations provide valid results in the context of reachability queries. In contrast to federated query approaches 3.3.4 where a source selection mechanism is required, the important characteristic of live exploration is that query execution do not require any *a priori* information about the distributed remote data sources being queried.

CHALLENGES AND LIMITATIONS OF LINKED TRAVERSAL APPROACHES Although Linked Traversal approaches seem to be fully compliant with the Linked Data principles, traversing live links may produce incomplete results. Furthermore, these approaches are not designed for high runtime performance [UHP+15].

3.3.7 Partial evaluation

The concept of partial evaluation has been used in many applications ranging from compiler optimization to distributed evaluation of functional programming languages [Jon96]. In recent years, partial evaluation has also been used for evaluating queries on distributed tree-structured data such as XML, and graphs [BCF+06a; CFK+12; FWW12; WWZ16b]. Authors in [BCF+06a; CFK+12] introduced the concepts of partial evaluation in XPath [CD+99] queries on distributed XML structured data. In their work, XPath queries are serialized to a vector of subqueries. Their approach finds partial answers for all subqueries at individual sites by using a top-down [CFK07] or bottom-up [BCF+06b] traversal, in a topological order, over the XML tree. Finally, all partial answers received from each site are assembled together at the server. However, in contrast to XML tree structured data, RDF data and query language (i.e., SPARQL) are based on graphs. It is not feasible to serialize SPARQL queries and traverse the RDF graph in a topological order.

Partial evaluation on graphs has been considered in prior works [FWW12; WWZ16b]. Wang et al. in [WWZ16b] proposed a method for answering regular path queries (RPQs) on large-scale RDF graphs using partial evaluation. They partitioned RDF data and distributed it in a cluster, and used a dynamic programming model to evaluate in parallel the local and partial results of the RPQ on each computing site. To assemble these partial answers they designed and implemented an *automata-based* algorithm to produce complete answers.

CHALLENGES AND LIMITATIONS IN PARTIAL EVALUATION–BASED APPROACHES Although partial evaluation is a well studied concept, in the context of SPARQL queries and Linked data, it has only recently been used. One of the most well-known challenges with partial evaluation is the large number of intermediate results. Moreover, complex queries suffer a performance bottleneck when evaluating on large-scale distributed RDF graphs. In the context of distributed paths, to assemble these partial paths into complete paths is also a challenge. In chapters 5, 6,

and 7 we provide heuristic-based approaches to assemble partial paths retrieved from remote endpoints.

3.3.8 Summary

When centralizing, the size of the data becomes a bottleneck in terms of data management, memory management, querying, and performance. Distributed approaches are a valid solution to overcome such issues. Different platforms and approaches, as described above, have been adopted in distributed data management and querying. For example, Hadoop-based MapReduce approaches [DG10; BWY17; PSH+11], partitioned-based approaches [GSM+14; HAR11; LLT+13], vertex-centric approaches [XWJ+18; WWX+19; NS16], query federation engines such as [SHH+11; HMZ+17; QL08b], and link traversal approaches [UHP+15; HP17] have shown interesting results. These are all used for large-scale distributed data processing, and each of these approaches have advantages and disadvantages. We have explained challenges and limitations of each approach previously. Hadoop-based, Partitioned-based, and Vertex-centric systems represent homogeneous distributed models. They work in a controlled environment where every distributed source or database must have the same software and configuration. Furthermore, the partition and distribution of the data must follow certain procedures or constraints, which is incompatible with Linked Data sources. We have highlighted homogeneous model of this type in Section 2.4.1 of Chapter 2.

In contrast to the homogeneous approaches, we in Chapter 7 propose an hybrid type of architecture where we try to minimize the restrictions in terms of configuration and data distribution. We introduced an *index-free* and *cache-assisted* source selection mechanism which improves the overall query performance for distributed paths. Our proposed distributed path query engine can be adopted as a federated path query engine with little efforts. We will discuss this in Chapter 7.

The peer-to-peer architecture is also used in homogeneous systems, which have been implemented in many RDF-based data storage and querying approaches [ACH+04; BHH+06; CF04; HHK05]. Although the peer-to-peer architecture is considered scalable, the drawback of this architecture is the performance issue for query requests, because the queries must be flooded in the network for a targeted resource [CB14]. Further, there is no mechanism for node or datasource selection in basic architectures.

To overcome message flooding issues and to improve the query performance, in Chapter 5, we introduce a peer-to-peer architecture that includes a source selection mechanism to reduce the communication cost and improve the query performance. Moreover, in contrast to the standard peer-to-peer architecture where data is distributed and replicated to each node in a controlled way, our approach does not require such data distribution constraints, and any size of heterogeneous data can be loaded into any node.

In heterogeneous models, query federation engines such as [SHH+11; HMZ+17; QL08b] were developed to query distributed RDF data. In the context of path queries, to the best of our knowledge, none of them have implemented federated paths. In Chapter 6 we introduce an *index-based* path query federation engine that computes the federated paths within heterogeneous environments.

4 FOUNDATION: EXTENDING TOP-K PATH QUERIES

This chapter is the first part of this thesis which presents work related to querying paths in centralized systems for RDF data. This chapter proposes an extension for Top-k Path Queries to SPARQL1.1 property paths. In other words, it reports on an extension of SPARQL 1.1 property paths that allows to compute and return the k shortest paths matching a property path expression between two nodes. The algorithm proposed and demonstrated for evaluation in this chapter is a relatively straightforward solution, but performs efficiently compared to tailored solutions in the literature (see Section 3.2 of Chapter 3) in practical use cases.

This chapter is organised as follows: Section 4.1 highlights the motivations for this work and Section 4.2 will present preliminaries. Section 4.3 discusses the syntax and semantics of the :topk function, Section 4.4 provides a detailed description of the bidirectional *breath-first* search algorithm used and Section 4.5 reports on the evaluation of the approach and shows that the proposed solution for path computation offers a very promising, performance tackling graphs with tens of millions of triples. Concluding remarks and a note on future work are offered in Section 4.6.

4.1 MOTIVATION

While SPARQL 1.1 introduced a form of path queries, as discussed in Section 2.3.2 of Chapter 2 this feature does not allow to retrieve or count paths. There is a history behind this, as the seminal paper by Perez et al. [ACP12b] in 2012 warned us not to "count beyond the Yottabyte", i.e the paper showed not only that the – at that time current – SPARQL engines implemented property paths in an inefficient manner, but also that a query language feature that allowed to return (the number of) *all* property paths would become infeasible even in relatively small graphs with a potentially double-exponential number of solutions. The existential semantics which the authors proposed to resolve this issue (and which became eventually a part of the final SPARQL 1.1 standard), still has its limitations in practice where path existence alone is not sufficient. For instance, in professional social networks, typically everybody is (somehow) transitively connected to anyone. We might be interested,instead, in the k closest connections.

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT * WHERE {dbpedia:bob foaf:knows+ dbpedia:alice}
```

Listing 6: Limitation example of a SPARQL Property Paths query.

The SPARQL property path query in Listing 6 would not help here. It would simply return an empty binding. Likewise, for most practical routing applications, the mere reachability test is not enough. On the other hand, enumerating or even simply counting all paths between two given points is not required either: Only the top most relevant (e.g., shortest) paths need to be found.

A particular interest for path queries comes from the bioinformatics domain where the volume of semantic data is constantly growing [CBF+16]. For instance, in the area of cancer genomics, experts often need to discover relevant associations between biological and genetic entities (such as diseases, drugs, genes, pathways, etc.) requiring efficient querying mechanisms [HQD15; HMR16]. It is typical in medical research that multiple genetic features, their effects, the diseases and treatments to those diseases are studied together, often in a larger context such as the medical history of the patient. One of the key challenges in cancer genomics – a cornerstone of precision medicine – is to discover gene-disease-drug associations which might be relevant for the development of new treatment methods. Such associations essentially correspond to paths in the semantic databases. However, SPARQL 1.1 does not provide adequate means for returning such paths.

Using our syntax, the three most promising connections could be obtained with the following query:

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?path WHERE{
        ?path ppfj:topk (dbpedia:bob dbpedia:alice 3 "foaf:knows+") }
```

Listing 7: Path query using the topk function.

4.2 PRELIMINARIES

The approach discussed in this chapter is experimented on RDF graph data. The preliminaries about RDF graph and relevant terminologies are described in Section 2.2.2 of Chapter 2. Further, this chapter focuses on finding paths in RDF graphs. Paths and relevant concepts are defined in Section 2.1.1 of Chapter 2. The path expression defined in this chapter is close to the SPARQL1.1 specification of property paths (see Section 2.3.2), except for the representation of inverse properties, which are not currently supported in this extension.[1] Specifically the following syntax is supported:

$$P := Q^*|Q^+|Q^?$$
$$Q := a|![a_1, ..., a_k]|(P/P)|(P|P)$$

Here Q denotes an expression without occurrence restrictions, the unary quantifiers $*, +$ and ? respectively denote an unrestricted number of occurrences, a single occurrence and at most a single occurrence of a respective pattern Q. $a \in I$ is an IRI representing a property, the set

1 Note that inverse edges with specially marked inverse edge labels could be added to any graph by at maximum doubling the number of edges, thus not changing the overall complexity of graph processing.

negation $![a_1, ..., a_k]$, for $a_1, ..., a_k \in I$ is satisfied by a single property $b \in I \backslash \{a_1, ..., a_k\}$. Finally, (P/P) defines a sequence of path expressions and $(P|P)$ stipulates that only one of the expressions on the left and on the right of $|$ need to be satisfied (or). Both binary operators are associative, so parenthesis are omitted in sequence of repeated binary operators of the same kind. The top k shortest paths problem is defined below:

Given an RDF graph $G, s \in I, t \in I \cup L$, an integer k, and a regular path expression p

Compute first k elements of $P_G(s,t)$ satisfying p

4.3 SYNTAX, SEMANTICS AND IMPLEMENTATION

This section provides the semantics and implementation details of our proposed approach. The overall goal of this approach is to provide an implementation that can be embedded into the existing open source SPARQL engines, in particular Jena ARQ [Teaa], without a need for changing any legacy code. Although not part of any official specification, Jena SPARQL extension interfaces can be seen as a *de facto* standard which we make use of.

The implementation provided in this chapter is a robust solution for the k shortest path problem and is based on *bi-directional* search: both the source and the target node of the paths need to be specified. The number of desired paths k defaults to 1, and the path pattern P defaults to the path pattern $(! : *)$:.[2] These arguments can either be omitted or should be bound to a constant of type integer and a string describing a path expression, respectively (see Listing 8).

The Jena ARQ framework allows SPARQL variables to occur at the input argument positions: the path function of the presented approach requires those variables to also occur elsewhere in the query to ensure that, at the point the topk function is called, all its input parameters are bound (analogous to the notion of safety [Unm89] in Datalog). Upon each call (for each binding of s and t, the topk function queries the RDF graph again and computes the shortest paths between s and t as explained in the next section. For each input tuple, there are up to k output values, each representing a path. The approach presented in this chapter encodes the found paths as strings which can then be parsed by the calling applications to extract the properties and resources involved.

The type of Jena ARQ extension mechanism catering for all the above desiderata is known as a property function. Syntactically, property functions use infix notation, appearing in the WHERE clause of SPARQL queries as triples: the function is in the predicate position (predicates are often called properties, hence the name "property function").

Both the values in the subject and in the object positions are passed to the function as arguments, whereby one of the two is meant to be the output and hence needs to contain an unbound variable. To pass multiple arguments, RDF lists need to be used. The predicate denoting functions can be distinguished by a dedicated namespace. The first option is to directly instruct ARQ which Java class to instantiate using a pseudo-url "java:<java.namespace>.". The SPARQL function name needs to coincide with the name of the Java class implementing it in this case, which is not always convenient. The other way is to use a special registry of IRIs that resolve

2 We assume that the default namespace URI ':' does not occur as a term in the dataset.

to property functions (such as ppfj:topk) shown in Listing 8 example. Such a registry is provided by ARQ. The downside of this approach is a slight increase of boilerplate code and, most importantly, the necessity to rebuild the calling Java program, which rules out this option for existing applications with dynamic queries like Jena Fuseki [Jena]. When the matching between the special predicate name and the Java function is established by ARQ, the function is called for each tuple of constants binding the variables occurring in the triple that represent the property function call.

4.4 ALGORITHM

This section of the chapter presents the core of our approach, i.e., the implementation of the topk function itself, which we solve with a relatively simple approach. Our starting point is a bidirectional breath first search algorithm (BFS). We extend the algorithm with path pruning based on path expressions, and provide an implementation which is easy to adapt to arbitrary graph models and incorporates the path search in full SPARQL via the extension function mentioned in the previous section. The listing of the extended path search algorithm can be found below as algorithm 4.1.

The algorithm maintains the sets f_f and f_b of resources (RDF nodes) called *frontiers:* before the $i-th$ iteration of the search procedure, f_f contains references to resources reachable from the source in exactly $\lfloor i/2 \rfloor$ steps, and f_b refers to resources reachable from the target node in exactly $\lfloor (i-1)/2 \rfloor$ steps. At each iteration, either the forward frontier f_f (odd i) or the backward frontier f_b (even i) is advanced. A resource α referenced by both frontiers before the iteration i belongs to a path of length $i-1$. Since BFS is used, all paths of the specified length are identified at the respective iteration of the algorithm. The algorithm maintains the set of paths from the two terminal nodes to the respective frontiers using linked lists, so that if two paths have a common prefix, this prefix is only represented once in the memory.

Thus, the actual data items stored in frontiers are $traversaledges(n, e, pr, \gamma)$ where n denotes the node, e is the incident edge via which this node has been reached, pr is the reference to the preceding traversal edge (np, ep, pr_p, γ_p) constructed at the previous advance step in the same direction (i.e. on the one before previous iteration). The forward and the backward frontiers are advanced interchangeably. The meaning of γ is explained below.

To account for the property path pattern P in the process of search, we convert it into a nondeterministic finite automaton (NFA) using the library dk.brics.automaton [Møl11] by Anders Møller. The implementation is based on character strings. Thus, in a preprocessing step (not shown in algorithm 4.1), we map each property mentioned in the path expression P to a unique character. Furthermore, a special character $\perp$ is reserved to represent properties not used in P: such properties are not distinguished by P, therefore for the admissibility w.r.t. P, all such properties can be represented by one and the same symbol. The overall size of P is limited by the number of Unicode characters, which is perfectly sufficient in practice.

To cater for path checking also in the backward search, a second automaton based on the inverse of the path expression P is used: to invert P in our property path language, it suffices to recursively reverse all sequences occurring in P, i.e., to replace every sequence $(P1/P2)$ with $(P2/P1)$. Longer sequences $(P_1/.../P_k)$ (which we allow by virtue of associativity of /) are inverted as $(P_k/.../P_1)$. Two NFAs, Af and Ab are obtained respectively from the path expression

P and its inverse $P\prime$. At the frontier advancement step, for a node n in a frontier tuple (n, e, pr, γ), we only follow those incident edges e of n which are not rejected by the respective automaton A in its step γ (e is an outgoing edge for the forward frontier and incoming for the backward one).

Algorithm 4.1: Bidirectional BFS with Pattern Enforcement via NFA

Procedure `BidirectionalBFS(`$G, start, target, k, P$`)`
  $sol \leftarrow \emptyset$; `/* solution is found */`
  $A_f \leftarrow$ Automation (P);
  $A_b \leftarrow$ Automation (inverse(P));
  $f_f \leftarrow (start, \text{null}, \text{null}, \gamma_{init}^{A_f})$;
  $f_b \leftarrow (target, \text{null}, \text{null}, \gamma_{init}^{A_b})$;
  $(f_{act}, A_{act}) \leftarrow (f_f, A_f)$;
  $(f_{pass}, A_{pass}) \leftarrow (f_b, A_b)$;
while $|sol| \leqslant k$ *and not both* f_f, f_b *stable* **do** `/* check until the queue is empty */`
  if ADVANCE(f_{act}, A_{act}) **then**
    $sol \leftarrow sol \cup \text{filter}(P, \text{JOIN}(f_f, f_b))$;
    else
      Mark f_{act} as stable;
  end
  swap$(f_{act}, A_{act}) \leftrightarrow (f_{act}, A_{act})$;
end
return sol;
Procedure `ADVANCE(`f, A`)`
  $f\prime \leftarrow \emptyset$;
  $inc = \begin{cases} successor, & \text{if } f \text{ is the forward frontier} \\ predecessor, & \text{otherwise} \end{cases}$
  for $(n, e, pr, \gamma) \in f$ **do**
    for $(e\prime, n\prime) \in inc(n)$ **do**
      $\gamma \leftarrow$ nextstate $(A, \gamma, e\prime)$ **if** $\gamma \neq$ reject **then**
        $f\prime \leftarrow f\prime \cup (n\prime, e\prime(n, e, pr, \gamma), \gamma\prime)$
      end
    end
  end
  if $f \neq \emptyset$ **then**
    Update the active frontier: $f \leftarrow f\prime$;
    else
      return fail;
  end
Function `JOIN(`f_f, f_b`)`
  $res \leftarrow \emptyset$;
  for $(n, e_1, pr_1, \gamma_1) \in f_f, (n, e_2, pr_2, \gamma_2) \in f_b$ **do**
    $res \leftarrow res \cup \text{trace}(n, e_1, pr_1).\text{trace}(n.e_2, pr_2)$;
  end
  return res;

4.5 EVALUATION

We experimented with two datasets from the DBpedia SPARQL Benchmark [ABK+07] used in the "Top-k Shortest Paths in large typed RDF Datasets Challenge"[3], which was part of the 13th European Semantic Web Conference in 2016. Our hardware setup was a quad-core Intel i5 desktop machine with 8GB RAM. The used datasets correspond to the 10% sample and the full benchmark datasets, which we respectively denote 0.1DB and 1DB. The data is freed from blank and untyped nodes. For evaluation, we rely on a compressed index representation called *Header, Dictionary, Triples* (HDT) (see Section 2.2.5 of Chapter 2) of this RDF data. Table 4.1 lists the total number of triples, distinct subjects (IRIs), predicates, and objects, as well as the number of IRIs that occur both in the subject and the object position in the dataset. It is clear from the dataset statistics that the number of such shared objects is relatively small, within one promille of the number of triples.

Our experiment is based on the queries of the ESWC'16 challenge, which were four distinct pairs of start and target IRIs and two parameters, namely the value of k and an additional path restriction present in half of the cases. All restrictions had the pattern (<property>/!:* | !:*/<property>) thus stipulating that admissible paths could either start or end with a specific property. All combinations of input parameters used in the challenge are given in Table 4.2. A list of queries used for this work are presented in Appendix B.

Listing 8 presents the query that computes the required paths for one of the tasks using our property function topk.

```
PREFIX : <http://dbpedia.org/property/>
PREFIX dbr : <http://dbpedia.org/resource/>
PREFIX ppf: <http ://sparql/pathfn#>

SELECT ?path WHERE{ ?path ppf:topk (dbr:Karl_W._Hofmann
dbr:Elliot_ Richardson 8088 ":predecessor/!:*|!:*/:predecessor" ) }
```

Listing 8: Path query using the topk function.

Table 4.1: Datasets statistics.

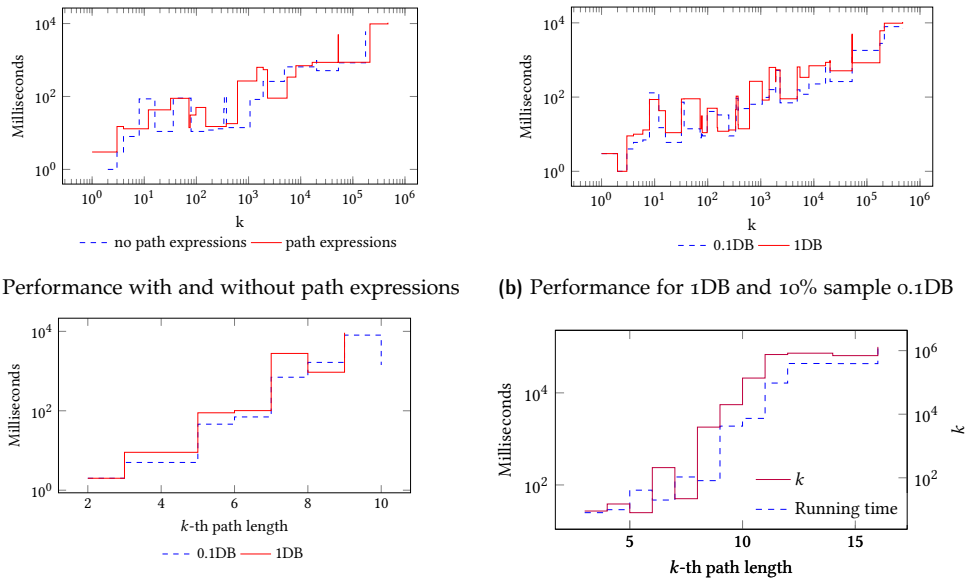
Data	Triples	Subject	Predicates	Objects	shared
0.1DB	9264609	313036	13114	3482820	58535
1DB	46275619	1457983	21875	13751780	462478

Table 4.2: Input parameters.

Start node	Target node	Filtered property
dbr:Felipe Massa	dbr:Red Bull	dbp:firstWin
dbr:1952 Winter Olympics	dbr:Elliot Richardson	dbp:after
dbr:Karl W. Hofmann	dbr:Elliot Richardson	dbp:predecessor
dbr:Karl K. Polk	dbr:Felix Grundy	dbp:president

We summarize the results of this experiment graphically in the following four figures. The first figure shows the dependency between the running time and the values of k used in each

3 cf. https://bitbucket.org/ipapadakis/eswc2016-challenge/downloads/



(a) Performance with and without path expressions

(b) Performance for 1DB and 10% sample 0.1DB

(c) Performance vs. the path length for 1DB and 10% sample 0.1DB

(d) Performance vs. random path length

Figure 4.1: Performance of the ESWC'16 challenge queries

task (queries). Note that different values of k are applied to different queries, so the chart is not monotone. However, one can see the trend of the running time increases with the growth of k. The two lines in Figure 4.1a show the average performance of the system running the queries based on Table 4.2, for the tasks with and without path expressions. Similarity as above, the dependency is not strictly monotone since not all queries are evaluated with the same values of k in the challenge. However, one can see that (i) the impact of path expressions on the performance in this particular case is negligible, and (ii) the dependency on k is linear (both axes in the figure are in logarithmic scale).

Figure 4.1b shows the performance of exactly the same set of queries run against the 0.1DB and the full DBpedia Benchmark dataset 1DB. It is worth mentioning that the intuition that the same queries would perform faster on smaller input is misleading for the top k paths queries. The reason is that the k^{th} shortest path computed on sampled data tends to be longer than if computed on the full dataset, since the same nodes might have a higher degree. For instance, the longest path for the sample is of length 10, and of length 9 for the full case. For 10 out of 38 queries of the challenge, the maximal path length on the 0.1DB was exceeding the maximal path length on 1DB by 20 to 25%. As a consequence, the search for k may take longer and thus the performance is sometimes worse on 0.1DB as Figure 4.1c shows.

In all cases, and on the HDT backend, our algorithm was able to compute every single path in a maximum of 9 and 10 seconds respectively for the 0.1DB and 1DB. The median running time is 46 ms, the average for the 0.1DB is 652 ms and for 1DB 929 ms. Each task with k smaller than 100 took at most 112 ms to solve.

We compared the performance of k shortest path queries over HDT with the SPARQL1.1 property path queries using the same source and target nodes, and the same path pattern. This allowed us to benchmark against popular SPARQL engines Blazegraph [Bla] (open source), Virtuoso Community Edition [Teab] (open source) and Stardog [Sir] (closed source, commercial). It has to be stressed that this is by no means a real benchmark, since our setup on top of an

HDT store is a read-only system highly optimized for querying, whereas the other three systems are full SPARQL engines with read and write functionalities. Furthermore, we made no performance optimizations beyond ensuring that the systems can use all available memory. The results of the comparison on 0.1DB are summarized in Table 4.3.

Table 4.3: Performance of top-k / reachability queries (in ms).

Query	k=1	k=100	Stardog (reachability)	Blazegraph, Virtuoso
Q1 (!:*)	24	94	57	N/A
Q1	125	6,138	19,732	N/A
Q2 (!:*)	15	132	45	N/A
Q2	19	2,568	34,707	N/A

Neither Blazegraph nor Virtuoso were able to cope with the queries on 0.1DB within our 16Gb main memory limit. Stardog was performing very well for unrestricted paths. However a path expression fixing any of the first or the last path edge resulted in a significant performance degradation. This suggests that an efficient implementation of regular expressions may be an important performance factor. One can see that for the value $k = 100$ our top k paths function stands well in terms of performance against the reachability test implementation of Stardog. Moreover, as Figure4.1a suggests, no clear performance penalty is associated with the necessity to enforce path patterns, due to our use of the BSD-licensed dk.brics.automaton library [Møl11]. The comparison also shows that property paths support in popular open source systems can be easily improved by using bidirectional search.

In the last experiment we searched for top k paths between nodes occurring at the ends of a random walk of a given length, which we varied in the range 5..20, in order to obtain the upper bound on the shortest path length. To also find long enough paths we were increasing the value of k. For 0.1DB, this approach typically allowed us to find paths of maximal length up to 14, in which case k was often close or even exceeding the value 106, since most of DBpedia paths belong to cycles, and have alternative paths in the graph (in particular, most properties with the http://dbpedia.org/propery namespace are also present in the namespace http://dbpedia.org/ontology which alone causes a combinatorial explosion of the number of paths).

Figure 4.1d shows that both k and the query evaluation time are exponential in the length of the top k-th path.

4.6 CONCLUSION

The plethora of interlinked data, as large-scale graphs, has created new challenges for data management and knowledge discovery. In our motivation, we highlighted that pathfinding, in the context of graph analysis, is an important property (particularly in bioinformatics domain, e.g., for finding associations between genes, diseases and drugs). However, with the growing amount of Linked Data, it has become a challenging task to perform such analytical task for large-scale graphs. In our problem statement (see Section 1.2 of Chapter 1) we briefly discussed that current SPARQL1.1 Property path queries are not sufficient to fully address these challenges. We also discussed the state-of-the-art-work and their limitations in the context of path queries (see Section 3.2 of Chapter 3).

To fulfill the research gap and to answer our first research question 1.3, in this chapter, we presented an efficient solution for the k shortest path problem in the context of SPARQL. Our solution is based on the adaptation of bidirectional breadth first search to top k paths computation, extending it with path expressions and embedding into Jena ARQ via the extension mechanism of property functions. We used indexed HDT backend data, and our implementation demonstrates very promising performance over large-scale graphs, even without complex optimizations.

Although, we demonstrated promising results for *centralized* RDF graphs, Linked Data by nature is distributed. Most of the approaches, including our approach, that work in a centralized way (i.e., only on single graphs) would require, to be applicable in this context, to merge all sources together, which can create issues of scale and in keeping the data up-to-date. To overcome these limitations, we extended our work for distributed paths. In the coming chapters 5, 6 and 7, we propose alternative solutions to distributed paradigms (see Section 2.4 of Chapter 2). In the next Chapter 5, we start with a *peer-to-peer* based distributed path computation model.

5 FEDS: DISTRIBUTED P2P PATH QUERY APPROACH

This chapter is the beginning of the second part of this thesis, which focuses on distributed approaches. We concluded previously in Chapter 4 that *centralized ways* of querying go against the distributed nature of Linked Data. Further, centralizing the data creates a bottleneck in terms of data management and memory management when the data is large, comes from multiple sources and needs to be updated. Consequently, distributed approaches have been considered in various works (see Section 3.3 of Chapter 3). This chapter presents our first approach towards a distributed system. We propose an approach – called FedS – that retrieves paths from multiple RDF triple stores. The key idea of this approach is to partially delegate the computational load to a set of distributed RDF triple stores in a *peer-to-peer* (P2P) manner, thus reducing the computational burden on a centralised query processing server. In our preliminary investigation, we evaluated and compare FedS against different solutions in terms of performance (overall path retrieval time) and result completeness, i.e., number of paths retrieved.

This chapter is organised as follows: Section 5.1 presents a motivation and a use-case scenario for this approach, and in Section 5.2 we highlight challenges in distributed environment. In Section 5.3 we define preliminaries. Section 5.4 briefly describes our approach, and Section 5.5 talks about results and discussion. Section 5.6 concludes this chapter.

5.1 MOTIVATION

The massive amount of data produced through scientific research (particularly in bioinformatics), corporate organisations, personal and social networks requires complex management techniques and tools, so to extract new knowledge out of it. For instance, in the biomedical domain, it is often the case that the data representations of two biological entities are scattered across different datasets distributed over some network. Looking at the issue of traversing paths in multiple graphs, the standard practice is to merge distinct graphs in a centralised way to evaluate the existence of paths between given entities (or nodes). As we discussed in the conclusion Section 4.6 of the previous chapter (4), centralising the data creates bottlenecks. In the context of Linked Data in particular, manually identifying which of the LOD 2.3 datasets contain the connecting properties is an impractical, cumbersome, and a time-consuming process.

5.1.1 Distribution and P2P

Distributed approaches for data management and querying purposes have been considered in many cases. One popular paradigm for distributed systems is to structure it through a P2P net-

work. Such networks offer the potential for low cost sharing of information, autonomy, privacy, and are considered to be scalable in homogeneous (see Section 2.4.1 of Chapter 2) distributed systems. P2P systems are usually built on distributed hash table (DHT) algorithms such as chord and pastry.

However, query processing in current P2P systems is very inefficient and does not scale well. The inefficiency arises because most P2P systems create a random overlay network where queries are blindly forwarded from node to node. Furthermore, the requirement for sophisticated distribution of the data over such networks is another issue. Generally, a loose guarantee of resource discovery in P2P network can make it possible not to find a resource although it does exist.

In this chapter, we provide a semi-structured P2P solution which not only improves the performance by reducing the network cost and guaranteeing source finding, but also does not rely on restrictive rules in terms of data distribution.

5.1.2 Use-case Cancer Genomics

In order to understand cancer progression, it is often the case that information about several genetic features, diseases and the medical history of the patient are studied together. One of the key challenges in cancer genomics is indeed to discover gene-disease-drug associations. Such novel associations provide insight into the drug development process tailored specifically for an individual patient (or a group of patients) targeting prevention, diagnosis and treatment of diseases [SR13b].

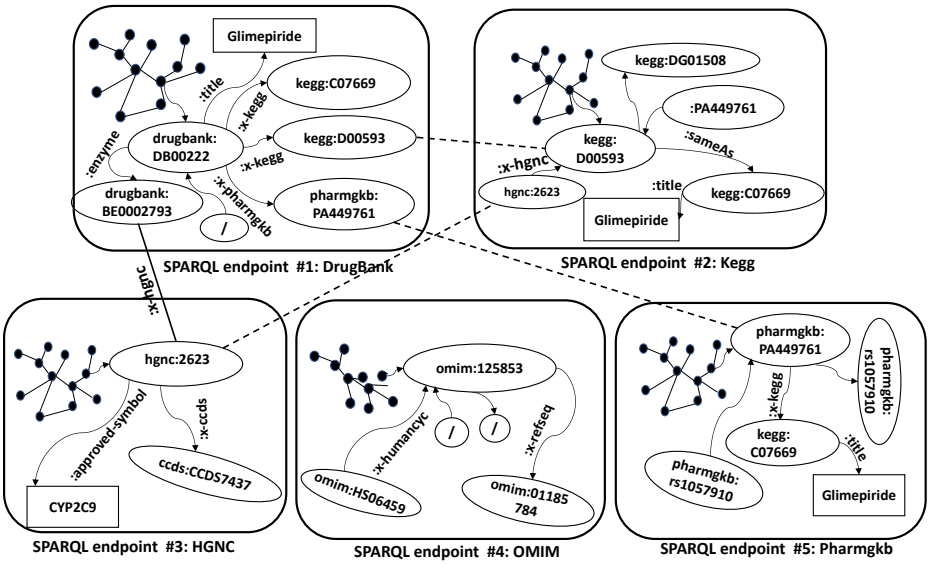


Figure 5.1: Datasets connectivity in the cancer genomics use case.

For instance, consider a scenario where a biomedical expert is trying to discover the paths between a drug `drugbank:DB00222` and a drug compound `kegg:C07669`. Figure 5.1 shows a group of datasets (DrugBank, KEGG, Hgnc, OMIM, and Pharmgkb) hosted at five different SPARQL endpoints. The source drug `drugbank:DB00222` is located in the SPARQL endpoint DrugBank, whereas the target drug compound `kegg:C07669` exists in the SPARQL endpoints 1 (DrugBank), 2 (KEGG), and 5 (PharmGKB). The source `drugbank:DB00222` – Glimepiride – is an antidiabetic drug whereas the target `kegg:C07669` represents a drug compound associated with Glimepiride.

If the biomedical expert needs to establish associations between the drug and its compound, s/he can find a direct relation by querying the SPARQL endpoint in DrugBank. However, for further analysis and to understand the mechanism of a drug compound and affected biological pathways, the expert needs to explore and discover associations in various others biomedical datasets. Figure 5.1 shows three paths starting from `drugbank:DB00222` (SPARQL endpoint 1) where the target `kegg:C07669` is available in (i) the DrugBank endpoint itself via the *:x-kegg* property; (ii) the Kegg endpoint via the *:x-kegg* and *:sameAs* properties; and (iii) the PharmGKB endpoint via the *:x-pharmgkb* and *:x-kegg* properties. The fourth path at the HGNC endpoint contains a node `hgnc:2623` via the *:enzyme* and *:x-hgnc* properties. However, the target node `kegg:C07669` does not exist in this endpoint.

Based on this example, a technology that can enable querying paths/associations among two or more biological entities across distributed repositories would be a great help to biologists and practitioners working in cancer genomics as well as in the larger healthcare and life sciences area.

5.2 CHALLENGES IN DISTRIBUTED ENVIRONMENT

Distributed environments also bring a number of challenges when it comes to querying and data management. We have discussed the challenges for different distributed approaches in Section 3.3 of Chapter 3. The important factor in efficient distributed approaches depends on design decision and what data is sufficient to search and gather the results from.

In the context of path associations, the distributed nature of the data causes paths to span over multiple datasets. There may also be cases where a single path can cross the same dataset many times. In contrast to the central environment where the whole path is always in a single graph, in distributed scenarios one dataset may include parts of the final paths in it, with the rest being distributed over multiple other datasets. This feature of distributed environments requires complex processes for merging partial results. In addition to this, there always exists a challenge with dividing and optimizing the computational process, taking into account the issue of tackling network costs.

5.3 PRELIMINARIES

The approach discussed in this chapter is experimented on RDF directed graphs. The preliminaries about directed graphs and their properties are discussed in Section 2.1 of Chapter 2. RDF graphs and their relevant terminologies are described in Section 2.2.2 of Chapter 2. Paths and relevant concepts are defined in Section 2.1.1 of Chapter 2.

In a federated environment, if a path between source and target vertices does not exist within a local single graph, other remote graphs are scanned and queried to find complete or partial paths contributing to the query results. In addition to the definitions already provided, we therefore also need to include the notion of source selection, as considered in our FedS approach.

Definition 8 ***(FedS Source Selection)*** *In a federated query environment, given a triple pattern* T *with subject* s_i *and object* o_i *in a query* Q *executed against a set of data sources* $\mathcal{D}$*, the set of relevant sources*

for T *in* $\mathcal{D}$ *is the set* $\mathcal{R}_t \subseteq \mathcal{D}$ *of data sources that can provide answers when queried with* T. *We use the notation* $\mathcal{R}_T$ *to denote the set of relevant data sources for source (subject) and target (object) nodes and use* $\mathcal{R}$ *when the context does not require specifying the triple patterns.*

For a given query to find the path between a source node s and a target node o, if these two are connected through any number of edges $p_i, \ldots, n$ we say that the path exists and (s,o) are reachable. If there is no path between s and o, it means that (s,o) are not reachable. We have defined *reachability* in Section 2.1.1 of Chapter 2.

5.4 FEDS

FedS approach works as a peer-to-peer (P2P) network of triple stores where every triple store implements the FedS algorithm (5.1). Figure 5.2 shows how the network of P2P datasets is organised as a fully connected set of federated systems. The FedS architecture is summarised in Figure 5.3, which shows its components: (i) Path Computation: path traversal starts from the host triple store where the path query is executed, (ii) Path Federation: during the local graph traversal, FedS also perform query federation towards the chosen endpoints selected by *source selection* on the fly, and finally (iii) Path Merger: paths retrieved from multiple triple stores are aggregated and the output as complete paths is produced.

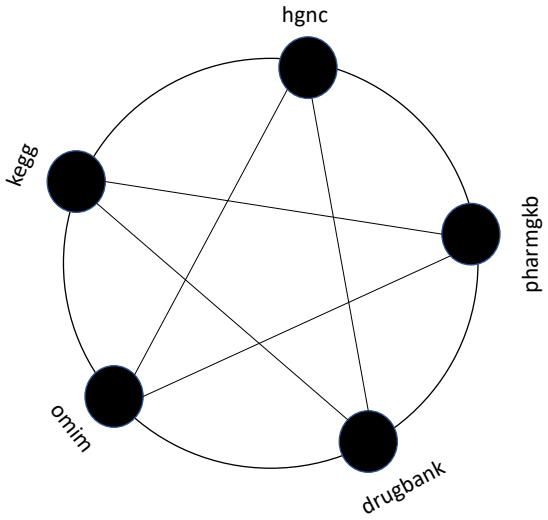


Figure 5.2: P2P network topology of endpoints.

5.4.1 Algorithm

The pseudocode of the main algorithm for FedS is shown in Algorithm 5.1. The queue q is a set of all paths starting at node *start* ordered by length. Every path extracted (Line 6) from the top of the queue q is extended by incoming edge e, of node n and a tuple tp. A tuple is a chain of nodes from start to the current node in the path. At Line 7, the leading node in path p is iteratively computed for its child n' linked through edge e' in graph G. If the child node (at Line 9) is a literal (i.e., not a URI) or has already been visited in that particular path p, the

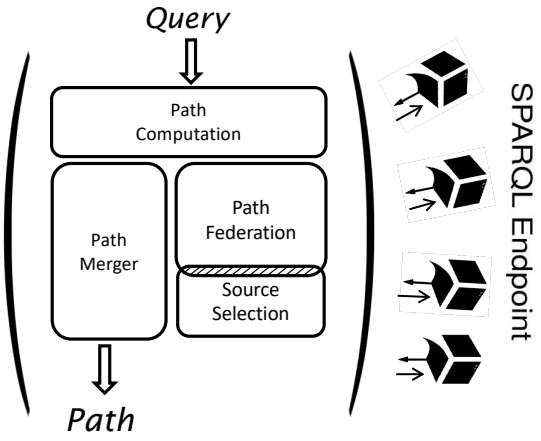


Figure 5.3: FedS Architecture.

algorithm does not consider it further and continues to the next iteration. It is important to note that checking visited nodes avoids cycles in a path and therefore possible infinite loops.

At Line 12, the existing path is concatenated with the traversed edge e' and child node n' to path tuple tp'. In the current iteration, the algorithm checks if the child node n' is *target*, and if so, the current tuple is added to the solution. The algorithm sets the flag doNotQueue true to avoid the addition of this path for next iteration in the q. If the target node is not found in the local graph G, then before iterating to the next child, the algorithm performs a test to check if the query was executed locally or the query was a remote request (Line 21).

If the query was executed locally, then an ASK query with the current node as source and the original target node is sent to remote datasets. On returning true from any remote endpoint, the algorithm constructs a path query and dispatches it to the selected triplestore. The retrieved results pr are added to the solution list (Lines 23 to 29). On the selected remote triplestore, an instance of the same algorithm will be be running and performing all the above steps. However, that triplestore only performs the local computation and returns the calculated paths to the endpoint from where the query was dispatched. Line 21 restricts sending queries to other endpoints if the request was sent from another endpoint. Line 38 of the algorithm adds those tuples to the queue where target node is not found. At the end if the solution size k is satisfied (lines 17 to 19), the algorithm is terminated.

In the following we discuss different components of FedS with a real-world use case where a user executes the following path query 9 to find k paths.

```
PREFIX drugbank: <http://bio2rdf.org/drugbank>
PREFIX kegg: <http://bio2rdf.org/kegg>
SELECT ?path WHERE{
        ?path ppfj:topk (drugbank:DB00222 kegg:C07669 3 ) }
```

Listing 9: Path query using the topk function.

Path Computation The traversal starts from the host triplestore in the P2P network. For a given query executed against the host triplestore, all the local (complete) paths are added to the solution list. For instance, assuming the host triplestore is the one containing the dataset

Algorithm 5.1: FedBFS algorithm's pesudocode

```
1  Procedure FedBFS(start, target, k)
2      sol← ∅ ;
3      𝒟 = {D1, ..., Dn} ;
4      q ← (start, null, null) ;                                  /* queue of tuples */
5      while q not empty do                                  /* check until queue is empty */
6          p ← (n, e, tp) ← q.poll() ; /* n is node, e is an incoming edge, and tp represents a
             tuple that is a tree from root to leaf node */
7          for edges e′(n, n′) ∈ G do ;                          /* iterate over local G */
8
9              if (n′ = literal) ∨ (isVisted(n′ ∈ p) = true) then ;          /* move next */
10
11                 continue;
12             tp′ ← p.concat(e′, n′)) ;
13             doNotQueue = false;
14             if n′ == target then
15                 doNotQueue = true;
16                 solutions ← solutions.add(tp′) ;
17                 if solutions.size ⩾ k then
18                     return solutions
19                 end
20             else
21                 if request.connection.localAddress== true then ;   /* if query request is
                     local */
22
23                     while 𝒟 not empty do          /* check until datasets set is empty */
24                         if ASK(n′, target) == true then ; /* check if current and target nodes
                             exist in remote dataset */
25
26                             pr ← FedRequest(n′, target,k) ;   /* a request to remote selected
                             dataset D */
27                             for p ∈ pr do ;       /* iterate each path in remote path list */
28
29                                 if p == complete then     /* if source and target in path is
                                 actual query requirement */
30                                     solutions.add(p) ; /* pr are paths from remote endpoint */
31                                 else
32                                     pm ← Path_Merger(tp′, p);
33                                     solutions.add(pm)
34                             end
35                         end
36                     end
37             if doNotQueue == false then                          /* if flag is false */
38                 q ← q.add(tp′)
39         end
40     end
```

Drugbank from Figure 5.1, Algorithm 5.1 finds a complete path `drugbank:DB00222` → :x-kegg → `keg:C07669` in the host triplestore and this path is added to the solutions. However, if any path from the source to target node is not reachable (see Section 2.1.1 of Chapter 2) within the host triple store, we defined it as a *partial path*. The partial path represents a chain of nodes from the source node to all its successors up to a leaf node, where the leaf node is not a target node. For instance, three simple partial paths in the host endpoint are:

i `drugbank:DB00222` → :x-kegg → `kegg:D00593`

ii `drugbank:DB00222` → :x-kegg → `pharmgkb:PA449761`

iii `drugbank:DB00222` → :enzyme → `drugbank:BE0002793`

In the path computation process each node in each individual path along with the actual target node (i.e., `keg:C07669`) is checked for all other triplestores in the P2P network. In the current example, the three partial paths from host endpoint lead to three different datasets, i.e., Kegg, PharmGKB, HNGC. However, only two datasets (i.e., Kegg and PharmGKB) participate where any node in the path along with the actual target node exist. The third endpoint (i.e., HGNC) will not be considered in the path as the target node is not present there. Further, the fifth endpoint (i.e., OMIM) is totally ignored as it was not selected by the source selection process.

Path Federation: Before federating path queries, FedS performs "source selection" of relevant triplestores (see definition 8). For this, it does not rely on an index of the content of those triplestores and only requires an index of the addresses of their query endpoints. Source selection is performed *on-the-fly* using the SPARQL ASK query that returns the availability of the two required nodes from different triplestores. After source selection is performed, the "path federation" component dispatches the request to the already selected endpoint. The traversal process starts in all those triple stores in the P2P network which receive the federated request and returns the retrieved paths towards the endpoint from where the request was dispatched.

Path Merger: This component on receiving the remote paths from different endpoints merges paths (both remote and local) where required. For example, for the given query in Listing 9 only two endpoints (i.e., Kegg, PharmGKB) participate in the results as they are the only ones containing the target node (i.e., `keg:C07669`). The following lists the paths returned from those two triplestores:

i `kegg:D00593` → :sameAs → `kegg:C07669`

ii `drugbank:PA449761` → :x-kegg → `kegg:C07669`

The path merger takes the local partial path and remote paths and merges them. The following lists the complete paths obtained after this component performed its tasks:

- `kegg:DB00222`→ :x-kegg→ `kegg:D00593`→ :sameAs→ `kegg:C07669`
- `kegg:DB00222`→ :x-kegg→ `drugbank:PA449761`→ :x-kegg→ `kegg:C07669`

5.5 RESULTS AND DISCUSSION

The experimental setup comprises of five datasets (i.e., SPARQL endpoints) and six input queries. We compared FedS against three systems (Virtuoso, Blazegraph, and HDT-Bidirectional-TopK).

Datasets: We downloaded data from release-4 of the Bio2RDF datasets[1]. The data cumulatively is around 3.89 GB, including DrugBank (size 1.18GB), KEGG (size 471.9MB), PharmaGKB (size 29.2MB), OMIM (size 1.61GB), and HGNC (size 592MB), with 22,363,257 triples, 2,343,770 subjects, 334 predicates and 8,159,944 objects. Table 5.1 shows the number of triples, subjects, predicates, and objects in each dataset and the hardware specs of five desktop machines used to conduct the experiments are shown in Table 5.2.

1 http://download.openbiocloud.org/release/4/

Table 5.1: Dataset statistics.

Dataset	Size	Triples	Subject	Predicates	Objects
Drugbank	1.18GB	5151714	421348	104	2472011
Kegg	479.1MB	3281579	358844	63	1835508
Pharmgkb	29.2MB	191379	19905	32	123336
Omim	1.61GB	9687186	1127394	93	1415364
HGNC	592MB	4051399	416279	42	2313725

Table 5.2: Specifications of virtual machines used in experiments.

OS	Data loaded	RAM	Hard disk	Processor
MAC	Omim	16GB	500GB	2.6 GHz Intel Core i5
Ubuntu	Drugbank	32GB	500GB	2.9 GHz Intel Core i7
Windows	Kegg	8GB	250GB	2.2 GHz Intel Core i7
Ubuntu	Hgnc	8GB	300GB	2.5 GHz Intel Core i5
Windows	Pharmgkb	16GB	250GB	2.2 GHz Intel Core i5

Queries: Table 5.3 shows six evaluation queries. Column two in Table 5.3 shows the number of datasets (SPARQL endpoints) contributed in path computation for each query. For example, in the case of **Q1**, DrugBank and Pharmgkb participated in retrieving complete paths. Similarly, in **Q5**, four endpoints (DrugBank, Kegg, HGNC, and Pharmgkb) contributed to the computation of different paths. The "Hops" column includes the number of properties (named edges) between source and target nodes (i.e., the path length).

Table 5.3: Six Input Queries and Results

Query	Dataset selected	Source	Target	Paths	Hops
Q1	Drugbank ⋈ Pharmgkb	drugbank:DB00072	clinicaltrials:NCT01959490	1	2
Q2	Drugbank ⋈ HGNC	drugbank:DB01268	hgnc.symbol:FLT3	1	3
Q3	Drugbank	drugbank:DB00134	kegg:D04983	1	2
	Drugbank ⋈ Kegg	drugbank:DB00134	kegg:D04983	1	1
Q4	Drugbank	drugbank:BE0002362	hgnc.symbol:CYP3A5	1	1
Q5	Drugbank	drugbank:DB00222	kegg:C07669	1	1
	Drugbank ⋈ Kegg	drugbank:DB00222	kegg:C07669	1	2
	Drugbank ⋈ Pharmgkb	drugbank:DB00222	kegg:C07669	1	2
	Drugbank ⋈ HGNC ⋈ Kegg	drugbank:DB00222	kegg:C07669	1	4
Q6	Drugbank	omim:147470	hgnc.symbol:IGF2	3	3x1

Performance Analysis: At the time of experimenting with this approach we could not find any distributed path query approaches that can work in a P2P environment and fit our scenario. Thus, we compare FedS against centralized stat-of-the-art triplestores (i.e., Virtuoso and Blazegraph) which support SPARQL1.1 Property Paths, and our previous work HDT-BidirectionalTopK. presented in Chapter 4. It is important to note that our aim in comparing FedS with HDT-BidirectionalTopK was not to assess the performance, but to cross-verify the number of total paths FedS retrieves. The reason we do not consider HDT-BidirectionalTopK for direct comparison is that, first, it is centralized and works on local graphs, and second, that it uses HDT 2.2.5 as underlying data model, which is compressed. Therefore, that approach is naturally faster than our distributed P2P approach.

To query and load data into local approaches, we created a single graph by merging data from the five considered datasets (see Table 5.1). On the other hand, we loaded individual graphs into five different systems (see Table 5.2) each running the Fuseki server [Jena] and an instance of

the FedS algorithm. We executed SPARQL 1.1 property path queries without specifying any predicate in the path expression (e.g., (: |! :)∗) against each approach.

Figure 5.4 shows a performance graph for all six queries executed against each approach. FedS, Virtuoso and Blazegraphs work on raw RDF data and therefore their results are comparable. The key point to be noted here is that FedS has to cope with additional communications delay over the network. Considering this, the results still appear promising. We notice also that Virtuoso and Blazegraph were not able to return results for **Q6**. In the case of Virtuoso, all queries were interrupted by a *time-out* error if the queries were executed without specifying the named graphs[2].

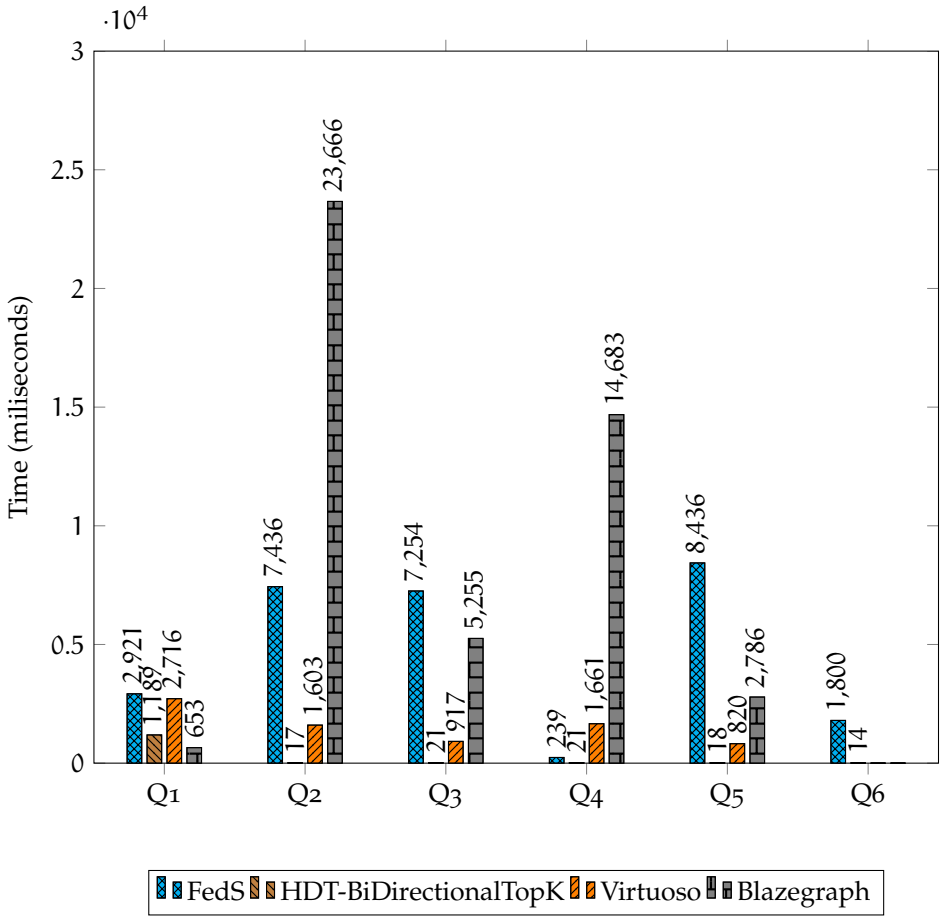


Figure 5.4: Comparing total query execution time.

Path Analysis: Table 5.4 shows a comparison of the *total number of paths* retrieved and their corresponding *path length* (i.e., Hops). For all the six queries with local approaches. FedS retrieved the same number of paths in a federated environment.

5.6 CONCLUSION

In this chapter we proposed FedS, a path traversal approach that works in a P2P network. The idea behind this work is motivated by the need of the biomedical domain to find associations

2 https://www.w3.org/TR/rdf-sparql-query/#namedGraphs

Table 5.4: Comparison of the Total Paths **#TP** and Path Hops **#PH**

	FedS		HDT-BiDirectionalTopK		Virtuoso		Blazegraph	
Query	#TP	#PH	#TP	#PH	#TP	#PH	#TP	#PH
Q1	1	2	1	2	1	2	1	2
Q2	1	3	1	3	1	3	1	3
Q3	1	2		2	1	2	1	2
	1	1	1	1	1	1	1	1
Q4	1	2	1	2	1	2	1	2
Q5	1	2	1	2	1	2	1	2
	1	2	1	2	1	2	1	2
	1	1	1	1	1	1	1	1
	1	4	1	4	1	4	1	4
Q6	3	3x1	3	3x1	-	-	-	-

(paths) across different biological entities. The current SPARQL 1.1 Property Path specification and the standard traversal algorithms (BFS, DFS, A*, etc.) assume (or require) a single graph – or many graphs merged into a centralised graph – for graph traversal. As we mentioned earlier, centralized approaches suffer various limitations and become a bottleneck in many cases where large-scale data is naturally distributed in multiple sources. FedS is a distributed P2P approach that not only tackle these limitations but also the existing drawbacks (highlighted in the motivation, Section 5.1) in P2P systems. Our initial evaluation results are encouraging, with FedS retrieving all the paths in comparable query processing times when compared to state-of-the-art triplestore approaches. However, in our current implementation FedS also has some limitations which are as follows:

- each SPARQL endpoint in the network has to maintain an index for the list of endpoints
- no cache was used so duplicated requests might be sent
- at the time of implementation, there was an assumption that the endpoint from where the query is executed contains the source node. There was no implementation provided to check and distribute the request if the source node did not exist in the local endpoint
- there might be complex partial paths where more than two datasets are involved. Such types of paths are called indirect paths. In the current implementation, we do not consider indirect paths. For instance, in Figure 6.1 Drugbank is linked with HGNC through `drugbank:DB00222`→ :anzyme → `drugbank:BE0002793` → x:hgnc → `hgnc:2623`. The HGNC endpoint does not contain the target node `kegg:C07669`. However, node `hgnc:2623` is leading to the Kegg endpoint where the target node `kegg:C07669` does exist

The approach we presented here does not follow the core P2P overlay network architecture or DHT algorithms (e.g., chord or pastry), and is also not bounded by restricted rules or sophisticated data distribution mechanisms as is required in standard P2P architectures. However, FedS belongs to the homogeneous networks paradigm (as discussed in Section 2.4.1 of Chapter 2) and works similarly to a distributed system in a controlled environment. As Linked Data by nature is distributed, in a fully controlled architecture, we may not be able to harness the full benefit of distribution. Consequently, in the next chapter (Chapter 6) we propose a federated path query

engine called QPPDS that can federate path queries to triplestores distributed over the Linked Data cloud and support the enumeration of paths for a given path query (such as Stardog [Sir]).

6 QPPDS: DISTRIBUTED INDEXED-BASED PATH QUERY APPROACH

This chapter presents our second approach in distributed path query processing. In this chapter, we propose an indexed-based path query engine – called QPPDs (Querying Property Paths Over Distributed RDF Datasets) – that computes paths from distributed SPARQL endpoints. QPPDs provides a heuristic-based source selection mechanism to select the relevant datasets (also called data sources) for a given path query, and a technique that federates queries to selected sources, and assembles (merges) the paths (partial, or complete) retrieved from those remote datasets. We demonstrate our approach on a genomics use case, where the description of biological entities (e.g., genes, diseases, drugs) are scattered across multiple datasets. We evaluate the QPPDs approach with real-world path queries on biological data, which are very heterogeneous in nature, in terms of performance (overall path retrieval time) and result completeness (number of paths retrieved).

This chapter is organised as follows: Section 6.1 presents the motivations behind this approach, and Section 6.2 highlights a genomics use case and also a toy example which makes it easier to understand the flow of the QPPDs algorithm. Section 6.3 gives a brief overview of QPPDs and discusses its core components. Section 6.4 presents the evaluations and results, and finally Section 6.5 concludes this chapter.

6.1 MOTIVATION

The previous chapter (5.1) talked about distributed path querying in a P2P architecture. We mentioned in Section 5.2 that P2P networks have their own limitations which we tried to address in that chapter. However, the approach we proposed belongs to the category of homogeneous systems (see Section 2.4.1) that work in a controlled environment. We also mentioned the limitations of our P2P approach in the conclusion section (5.6). Considering those issues and limitations in the P2P architecture, such systems are not fully aligned with the natural distribution in Linked Data over heterogeneous datasets (see LOD cloud 2.3).

Standard SPARQL query federation was introduced in SPARQL1.1 through the SERVICE clause (see Section 2.4.2 of Chapter 2) which allows us to send subqueries to remote endpoints in a federated manner. Extensive work (see Section 3.3.4 of Chapter 3) was also done, beyond SPARQL 1.1, on SAPRQL query federation. However, surprisingly, none of the already proposed query federation approaches support *distributed path querying*.

Our third motivation is related to the need to generate a dataset of paths that could include provenance and statistical information for each path (*full or partial*) retrieved for a given query. This would include the number of datasets participating to compute each path, what portion of

a particular path is retrieved from which endpoint, etc. Existing systems and approaches both for distributed and centralised architectures discussed in Chapter 3 do not provide such type of datasets which include provenance information.

6.2 MOTIVATING SCENARIO

In this section, we present two motivating scenarios: (1) a real-world scenario showing the use of distributed property paths in RDF datasets for Cancer Genomics; and (2) a toy scenario which is used as a running example to explain the proposed approach.

CANCER GENOMICS The background and context of this use case is similar to the one we presented in chapter 5.1. However, the datasets we used in this approach are different.

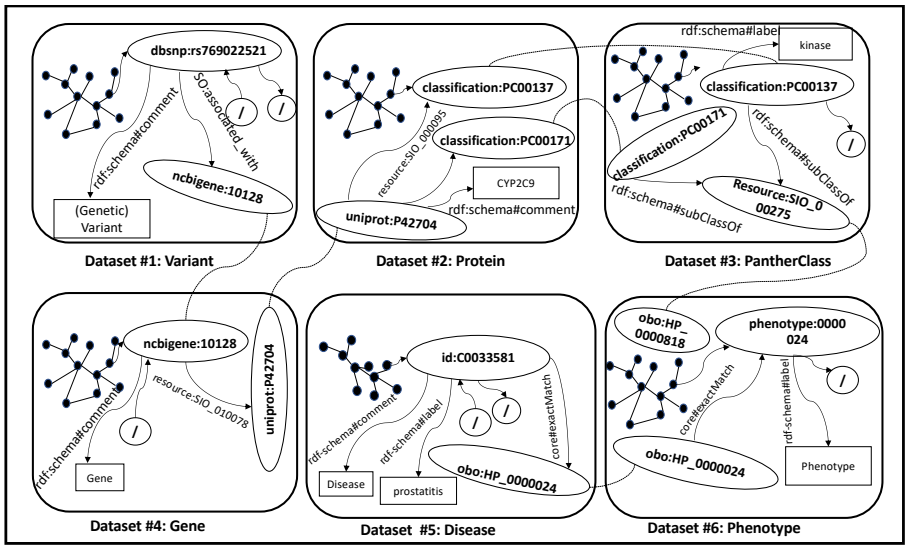


Figure 6.1: A real-world example of distributed paths between RDF datasets.

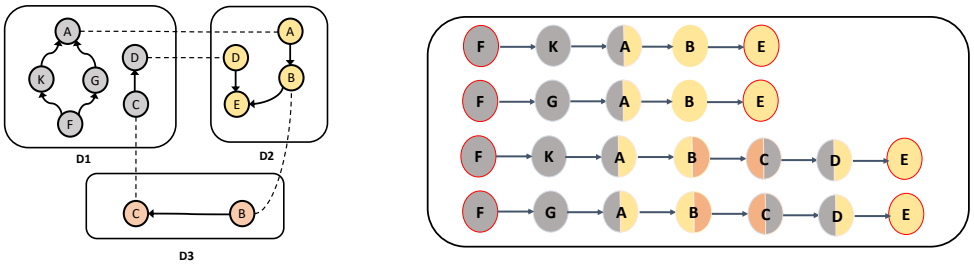
For instance, consider a scenario where a biomedical expert is trying to discover the paths between a gene `rs769022521` and the associated disease `HP_0000024`, and (s)he only has access to a single local dataset (e.g., Variant, see Figure 6.1). Even if those two entities are present in that dataset, existing paths between them might not be, in which case, the expert would not be able to find the association between `rs769022521` and `HP_0000024`, unless (s)he does have access and knowledge about other datasets. Even if that was the case, however, without an approach to identify paths across those datasets, the expert would have a hard time finding paths without first integrating all datasets into one.

Figure 6.1 shows how a path between `rs769022521` and `HP_0000024` can be obtained where each of the datasets i.e., (Variant, Gene, Protein, PantherClass, Phenotype, and Disease) hosted at different SPARQL endpoints contributes to this particular path.

Based on the above scenario we believe that a technology –implemented here through the QPPDs approach– that can enable querying paths/associations among two or more biological entities across distributed datasets/endpoints would be of great help to biologists and practition-

ers working in cancer genomics as well as in the larger healthcare and life sciences area. This is further demonstrated in the evaluation section of this chapter.

RUNNING EXAMPLE In this scenario, we present a toy/fictional example, used in the rest of the chapter to illustrate the approach. It uses three synthetic RDF datasets given in Figure 6.2a. The goal of presenting this use case is to explain the proposed approach by using a simple and easy to follow example. Suppose that we want to find all the paths from resource F of Dataset D1 to resource E which is present in Dataset D2. A simple SPARQL property path query on each dataset individually will not be able to retrieve any path. This is because both source F and target E nodes are not present within the same dataset in any of the datasets. However, we can obtain partial paths from different datasets which can later be assembled or re-arranged to form complete paths between the two distributed resources as given in Figure 6.2b.



(a) Sample datasets and their connectivity.

(b) Federated Paths.

Figure 6.2: Running example: Datasets and paths between node F and node E.

6.3 THE QPPDS APPROACH

The QPPDs engine consists of four main components: (1) the QPPDs index for distributed path computation, (2) path computation between the connected datasets, (3) distributed path computation between the resources within the connected datasets, and finally (4) the path merger. Figure 6.3 depicts the architectural diagram of QPPDs. We explain these components in the next sub-sections with the help of our running example and walk through each step in QPPDs.

6.3.1 The QPPDs Index

The QPPDs approach makes use of a pre-computed index for fast retrieval of K possible paths between the source and target resources in distributed RDF graphs. We assume a set of data sources $\mathcal{D} \coloneqq \{D_1, \ldots D_n\}$ where each data source $D \in \mathcal{D}$ is a SPARQL endpoint. We say a connection $\Psi(D, D')$ holds between two datasets D and D' if both datasets have a common node, i.e., $\{\Psi(D, D') \mid \exists N : N \in \mathcal{D} \wedge N \in \mathcal{D}'\}$. Furthermore, $Rs(D, D')$ represents all such common nodes between $D \wedge D' \in \mathcal{D}$. For each data source $D \in \mathcal{D}$, QPPDs stores the following as index:

1. The set of datasets connected to D: $Cs(D) \coloneqq \forall_{D' \in \mathcal{D}}\{D' \mid \Psi(D, D') \wedge D \neq D'\}$ (`:connectedTo`).

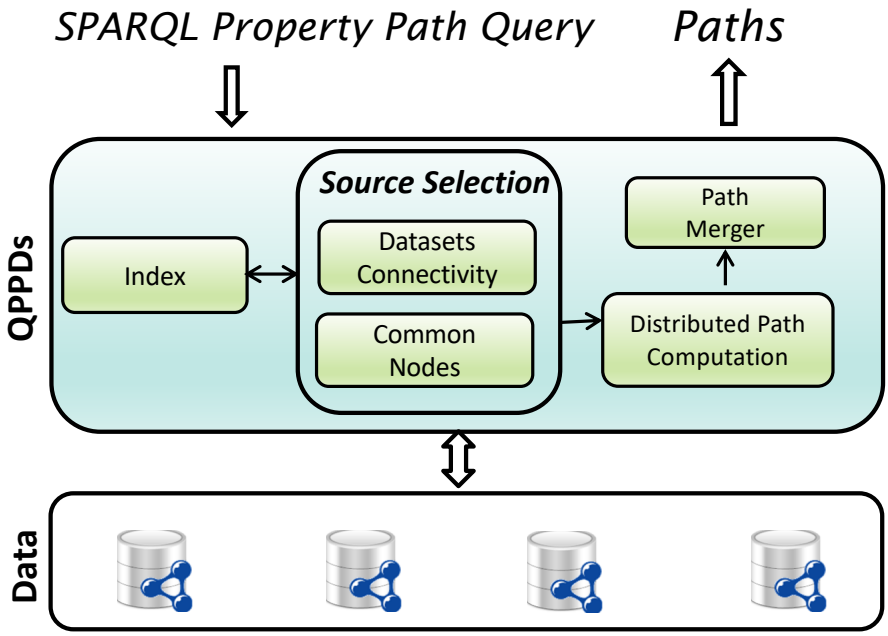


Figure 6.3: QPPDs architecture.

2. For each dataset $D' \in Cs(D)$ the set of resources which connects (i.e., common) D and D', i .e., $Rs(D, D')$ (`:isCommonIn`).

Listing 10 shows an excerpt of the corresponding index of the datasets of the example given in Figure 6.2a. Please note that we used a simplified representation for the sake of simplicity. In reality, our index is represented as valid RDF in the NTriples format. We compute the index of the datasets by simply sending relevant SPARQL queries to the corresponding endpoints of the datasets.

```
## ConnectedTo --------
    D1  :connectedTo   D2,D3 .
    D2  :connectedTo   D1,D3 .
    D3  :connectedTo   D1,D2.
## isCommonIn --------
    A   :isCommonIn    D1, D2 .
    B   :isCommonIn    D2, D3 .
    C   :isCommonIn    D1, D3 .
    D   :isCommonIn    D1, D2 .
```

Listing 10: The QPPDs index of the datasets of motivating example given in Figure 6.2a.

Algorithm 6.1 shows the QPPDs index generation process, which takes the set of all datasets $\mathcal{D}$ as input and returns the corresponding QPPDs index I as output. For each dataset $D \in \mathcal{D}$, we first get the set of all URI nodes (i.e., resources) $\mathcal{N}$ (Lines 1-2 of Algorithm 6.1). The set of all URI nodes from a dataset are retrieved by using a single SPARQL `SELECT` query given in Listing 11. For each node $N \in \mathcal{N}$, we then create a SPARQL `ASK` query (given in Listing 12) and send it to all datasets D (Lines 3-5 of Algorithm 6.1). The datasets that return true to the given SPARQL `ASK`

are treated as connected to D and share a common node N. All common nodes are added to the index (Lines 6-9 of Algorithm 6.1).

Algorithm 6.1: QPPDs index construction

```
   input  : 𝒟 ≔ {D1,...Dn};                          /* All datasets */
   output: I;                                       /* The QPPDs index */
 1 foreach D ∈ 𝒟 do
 2 |  𝒩 ← getURINodes(D);
 3 |  foreach N ∈ 𝒩 do
 4 |  |  foreach D' ∈ 𝒟 ∧ D' ≠ D do
 5 |  |  |  b ← ASK(N,D');
 6 |  |  |  if (b == 'true') then
 7 |  |  |  |  I.add(D : connectedTo D');
 8 |  |  |  |  I.add(N : isCommonIn D');
 9 |  |  |  end
10 |  |  end
11 |  end
12 end
13 return I
```

```
SELECT  ?s ?o
WHERE   { ?s  ?p  ?o. FILTER isURI(?o)  }
```

Listing 11: SPARQL query to select all subjects and objects.

```
ASK WHERE { ?s ?p ?o. Filter(?s = <%N> || ?o= <%N> ) }
```

Listing 12: SPARQL query to ASK subjects and objects.

6.3.2 Paths Computation between Datasets

In Linked Data, RDF datasets are interconnected via various links. Considering our motivating example given in Figure 6.2a, all three datasets are interconnected via different links. In order to compute paths – for a given query (see Listing 14) – between resources within the distributed datasets (Algorithm 6.3), we first need to compute all possible paths to reach from one dataset to another dataset. For example, in our synthetic motivating example, the possible paths between Dataset D1 and Dataset D2 (under the condition that the maximum allowed repetition of nodes in a path equals 2) are given in Figure 6.5. Thus, to compute all possible paths between two datasets, we first need to make a graph of datasets similar to Figure 6.4 and then apply some algorithm on this graph to retrieve the required paths. We make use of the index (i.e Listing 10) to get the required graph of datasets and then apply a version of Breadth-First-Search (BFS) with some modifications to calculate the paths over the given graph. Algorithm 6.2 explains how the paths are calculated from the source and target datasets in a multi-graph of interconnected datasets.

Algorithm 6.2 finds k paths between the source and target nodes in multi-graph. Thus, it requires the source node s (source dataset in our case), target node t, number of paths k, and multi-graph G as input and retrieves the top-k shortest paths between the source and target nodes as output. Lines 6-11 correspond to a standard BFS-based approach. However, at Line 12 we slightly diverge from BFS: During the traversal, if a node is already visited, we do not consider it as a visited node, but this visited node is again queued to traverse until it reaches the maximum visited count of the maximum links between any two nodes in G. For example, the maximum links between two nodes in the multi-graph given in Figure 6.4 is 2. Using this modification, we are able to retrieve paths involving cycles in the given graph. The rest of the algorithm corresponds again to a standard BFS approach where queue q stores all the paths starting from the start D_{source} (e.g. D1) in Figure 6.5 ordered by path length. In every next iteration, the path p is extracted from the queue q extending p by one hop edge. The new extended path p' along with the previous path is again queued and whenever the next extended edge leads to the D_{target}, (i.e. D2) in Figure 6.5, this complete path is stored in the list of solutions sol (Line 18 of Algorithm 6.2). When the number of solutions reaches k, the traversing process is terminated.

Algorithm 6.2: Algorithm to find k paths between source and target datasets

```
1  s← D_source ;                                                    /* source node */
2  t← D_target ;                                                    /* target node */
3  k← K_paths ;                                          /* search number of TopK paths */
4  D← G ;                  /* A multi graph generated by CONSTRUCT query over index */
5  sol← ∅ ;                                             /* solution (retrieved properties) */
6  q[∅] ← |T_i,...,T_n| ;                                  /* queue of triples T(s,p,o) */
7  while (D_i)_{i=1}^n ≠ ∅ do
8      q← {T(s,p,o)} ;          /* pull and remove triple(T) with source node into queue */
9      while q ≠ ∅ do                                    /* check until the queue is empty */
10         (tp_i)_{i=1}^n ← q.T(s,p,o) ; /* get all child nodes of source node(s) and corresponding
            triples(tp) */
11         for (tp_i)_{i=1}^n do
12             if (tp(o)_i = literal) ∨ (isVisted(tp(o)_i) >= maxNodesLinks) then ; /* move next */
13
14                 continue;
15             else if (tp(o)_i = t) then;                          /* if target node found */
16
17                 p← path(tp(o)_i) ;                                        /* store path */
18                 sol← p ;                                        /* solution is found */
19                 flag← true ;          /* do not store in the queue since path is found */
20                 if sol.size ⩾K then
21                     return sol;
22                 end
23                 else if flag ← false then
24                     q← q.add(tp(o)_i)
25             return sol ;                                          /* return solution */
26         end
27     end
28 end
```

6.3.3 Distributed Path Computation

In this section we explain, in detail, the QPPDs distributed path computation given in Algorithm 6.3. The algorithm takes the source node n_{source} and target node n_{target} for which

all paths are required to be computed over the set of distributed datasets $\mathcal{D}$. The algorithm –incorporating with previous algorithms 6.1 and 6.2– retrieves all distributed paths between the required two nodes.

The source and target datasets (i.e., D_{source} and D_{target}, respectively) at step 1 are selected, by sending SPARQL `ASK` queries to all of the datasets $\mathcal{D}$, before the actual path query in Listing 14 for n_{source} and n_{target} is executed. The datasets which return true for n_{source} are added to set S and datasets which return true for n_{target} are added to set T (Lines 2-5 of Algorithm 6.3). In our running example, n_{source} is F and n_{target} is E. Since F can only be found in D1, hence $S \coloneqq \{D1\}$ and E is only present in D2, therefore, $T \coloneqq \{D2\}$.

In step 2, we need to compute possible paths from source datasets S to target datasets T within the generated index. To do so, we compute the multi-graph G, showing how the datasets in S are connected to the datasets in T (Line 7 of Algorithm 6.3). This graph can be constructed from the QPPDs index by using a SPARQL `CONSTRUCT` query given in Listing 13. Figure 6.4 shows the corresponding multi-graph of the datasets used in our running example. Now the graph G can be considered as a single RDF dataset and Algorithm 6.2 is used to find paths within the graph G. To this end, we run Algorithm 6.2 over graph G to find all paths from all the datasets in S to all the datasets in T (Lines 8-12 of Algorithm 6.3). In our running example, since $S \coloneqq \{D1\}$ and $T \coloneqq \{D2\}$, we need to find all possible paths from D1 to D2 in the multi-graph G given in Figure 6.4. Since G is a multi-graph with cycles, it is possible that we can have an infinite number of possible paths. However, according to graph theory, we can still retrieve complete paths while allowing a maximum number of repetitions of a node within a path equal to the maximum number of edges between two nodes in graph G. In the multi-graph given in Figure 6.4, the maximum number of edges between any two datasets is 2 (i.e., between D1 and D2). Under this condition, the paths retrieved by Algorithm 6.2 over the graph of Figure 6.4 are given in Figure 6.5.

```
PREFIX  feds: <http://vocab.org.centre.insight/feds#>
CONSTRUCT
{ ?s feds:connectedTo ?o . }
    WHERE { ?s  feds:connectedTo  ?o }
```

Listing 13: SPARQL CONSTRUCT query to find the :connectedTo

```
PREFIX ppfj: <java:org.centre.insight.property.path.>
SELECT * WHERE{
        ?path ppfj:topk (<source> <target> k) }
```

Listing 14: QPPDs SPARQL path query

Until step 2 of Algorithm 6.3, we have computed all paths from the source datasets (containing resource n_{source}) to target datasets (containing resource n_{target}). However, these paths are only at the dataset level, i.e., linking a dataset to another dataset, and do not tell us how to go from a node to another node within the distributed datasets. We achieve this in step 3 of the algorithm. In step 3, we follow dataset paths computed in the previous step to get actual paths at node-level.

Algorithm 6.3: QPPDs distributed property paths finding algorithm.

```
   input : Q = n_source ∧ n_target, 𝒟 := {D_1,... D_n};        /* query, set of all datasets */
   output: P ;                                                 /* Set of distributed paths  */
1  /* Step 1: get source and target datasets */
2  foreach D∈ 𝒟 do
3  |  S ← D_source + ASK(n_source,D) ;
4  |  T ← D_target + ASK(n_target,D) ;
5  end
6  /* Step 2: get all paths from each source to target datasets */
7  G ← getConnectedGraph(S,T) ;
8  foreach s ∈ S do
9  |  foreach t ∈ T do
10 |  |  SP ← SP + getPaths(G,s,t)
11 |  end
12 end
13 /* Step 3: get distributed paths from source resource to target resource */
14 foreach p ∈ SP do
15 |  pathLength ← p.length ;
16 |  count ← 0 ;
17 |  sources [] ← Ø ;                                        /* array of source resources  */
18 |  curDataset ← p.getNextDataset() ;
19 |  sources.addResource(n_source);
20 |  while (count < pathLength) do
21 |  |  nextDataset ← p.getNextDataset() ;
22 |  |  if (nextDataset ≠ Ø) then
23 |  |  |  targets [] ← getCommonIn(curDataset, nextDataset);
24 |  |  |  sources []=ifNodeHasChild (sources, curDataset);
25 |  |  |  targets []= ifNodeHasParent (targets, curDataset);
26 |  |  |  /* These nested loops work in batch style ;
27 |  |  |  and generate a query accordingly */
28 |  |  |  foreach source in sources [] do
29 |  |  |  |  foreach target in targets [] do
30 |  |  |  |  |  Q ← generateQuery (source, target);
31 |  |  |  |  |  P' ← FedRequest(curDataset,Q) ;
32 |  |  |  |  end
33 |  |  |  end
34 |  |  |  curDataset ← nextDataset ;
35 |  |  |  sources [] ← targets [];
36 |  |  end
37 |  |  else
38 |  |  |  target ← n_target ;
39 |  |  |  foreach source in sources [] do
40 |  |  |  |  Q ← generateQuery (source, target);
41 |  |  |  |  P'← FedRequest(curDataset,Q) ;
42 |  |  |  end
43 |  |  end
44 |  |  count++;
45 |  end
46 |  return P ← mergePaths(P');
47 end
```

We maintain a list of all the dataset paths $p \in SP$ calculated bewteen D_{source} and D_{target}. For each path $p \in SP$, we first compute its length, i.e. the number of datasets to be tested for query $Q = n_{source}, n_{target}$ (Line 15 of Algorithm 6.3). For example, in Figure 6.5 the first dataset path p1 is D1 → D2 with length 2. We then initialize a counter, an array of source resources, get the first dataset from the path, and add n_{source} (i.e., F) into the array of source

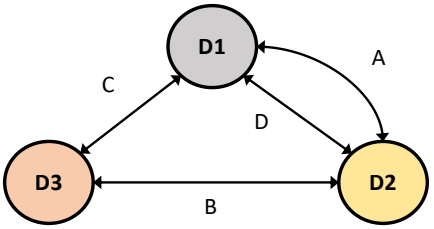


Figure 6.4: Multi-graph of datasets connectivity of our running example.

```
p1: D1 → D2
p2: D1 → D3 → D2
p3: D1 → D2 → D1 → D2
p4: D1 → D2 → D3 → D2
p5: D1 → D3 → D1 → D2
p6: D1 → D2 → D1 → D3→ D2
p7: D1 → D2 → D3 → D1→ D2
p8: D1 → D3 → D2 → D1→ D2
```

Figure 6.5: Possible paths from D1 to D2 in multi-graph given in Figure 6.4 with a condition of maximum allowed repetition of node in a path equals to 2.

resources. For p1 of our running example, curDataset would be D1 and resource F would be added into the sources array. Now at Line 20, we check if the count is less than path length, which is true in the current state of our running example (i.e., count = 0, pathLenghty = 2). We get the next dataset (D2 at current) from the path and store in nextDataset. Since the next dataset is not empty, we get all the common nodes in curDataset and nextDataset (Lines 21-23 of Algorithm 6.3) from the index. In our running example at present, the common nodes between D1 and D2 are A, D and these common nodes are added in to the target set. The common nodes between two datasets are retrieved by using the query given in Listing 15.

```
SELECT DISTINCT ?node WHERE {
    ?node :isCommonIn <%curDataset> .
    ?node :isCommonIn <%nextDataset> .
}
```

Listing 15: SPARQL query to find the common nodes between datasets

Walking through the running example, when we have two sets sources = {F}, targets = {A, D} and the dataset curDataset = D1 as well as the nextDataset = D2 (Lines 18-21 of Algorithm 6.3), we perform three steps to optimize our approach, just before sending the remote requests to curDataset = D1.

1. We check the set sources for all its elements (i.e., nodes) in the curDataset = D1 to see if they have children (Line 24 of Algorithm 6.3). After filtering out all nodes with no children, we get an updated set of sources. We do this because a path search should stop at a node having no further children. At the current stage of our running example, the sources set contains F, which has children, so we add it to the query generation.

2. Similarly, we filter out the targets set for the curDataset = D1 and discard all target nodes who do not have parent nodes except n_{target} (Line 25 of Algorithm 6.3). At the

current stage of our running example, nodes $\{A, D\}$ in set $targets$ both have parents and are therefore not to be discarded.

3. We also check if $n_{source} = n_{target}$. If true we do not count this combination in path search, because this points to the same node.

In our running example, after applying the first two optimization techniques, we still maintain $sources = \{F\}, targets = \{A, D\}$, however, it is important to note that it would not always be the case in real-life data. From Line 28-32 of Algorithm 6.3, nested loops for $sources$ and $target$ respectively iterate in a batch style (depending on user settings) and generate the nested SPARQL path query Q while considering the third optimization technique (Line 30 of Algorithm 6.3). We discuss the impact of a batch-based nested query in the results section. In our running example, the generated SPARQL query (see Listing 14) is sent to the $curDataset = D1$. The results P', i.e., $\{F \rightarrow K \rightarrow A\}, \{F \rightarrow G \rightarrow A\}$ retrieved from D1 are stored against this dataset. It is important to note that P' represents the *partial paths*. However, it does not mean that partial paths set P' will maintain always incomplete paths, but in real-world scenarios there may be many cases of complete paths retrieved and stored in P'.

When the iteration for $curDataset = D1$ is finished and the next dataset (i.e., D2) becomes the $curDataset = D2$ and set of $targets$ (i.e., $\{A, D\}$) becomes the $sources$ (Lines 4-35 of Algorithm 6.3), the procedure jumps back to Line 21 of Algorithm 6.3, where it checks if more datasets are available. In the current scenario of p1: (see Figure 6.5) D2 is the last dataset, therefore, control goes to Line 39 of Algorithm 6.3.

Now at Line 39, we get $sources$ (i.e. $\{A, D\}$) and the $targets$ set will only contain the actual target resource n_{target} since there is no further dataset to explore, which means no more common node needs to be checked. From Lines 39-42 of Algorithm 6.3, the query is generated for each element of the loop as a source, while keeping the target node static –i.e., the actual target node E of the query executed by the user– and sent to $curDataset = D2$. we get results P', i.e., $\{A \rightarrow B \rightarrow E\}$, which are stored against D2. At this stage, the length of $count$ reaches to the maximum, therefore the processing of p1: (see Figure 6.5) is terminated and control goes to Line 46 of Algorithm 6.3.

Now at Line 46 of Algorithm 6.3, mergePaths (Algorithm 6.4) is called, which merges all the remote paths, $P'\{F \rightarrow K \rightarrow A\}, \{F \rightarrow G \rightarrow A\}$ from D1 and $\{A \rightarrow B \rightarrow E\}$ from D2, to complete full paths P. After merging the paths we get a set of; $P\{F \rightarrow G \rightarrow A \rightarrow B \rightarrow E\}, \{F \rightarrow K \rightarrow A \rightarrow B \rightarrow E\}$.

So far, we have obtained the paths that are direct paths, meaning one-to-one relations between two connected datasets and no third dataset contributes in path computation. However, there may be many cases where indirect paths exist as shown in Figure 6.2b, where paths *#3 and #4* are *indirect paths*.

We step through in the following paragraph to understand how Algorithm 6.3 works when it encounters indirect paths.

INDIRECT PATHS : In our running example, we take path #3 i.e., $(F \rightarrow K \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow E)$. Unlike the previous paths discussed before, which involve only two datasets (i.e. D1 and D2), the indirect path involves three datasets (i.e. D1, D2 and D3) to complete it between F and E.

Having provided a detailed explanation before, we here provide only a simplified description.

This particular path #3 is calculated when Line 10 of Algorithm 6.3 faces p7: D1 → D2 → D3 → D1 → D2 dataset path connectivity given in Figure 6.5. We can notice here that the first two datasets of p7: i.e., D1 → D2, have already been explained for p1:. For simplicity, we skip their computation since we know that D1 and D2 return P' {F → K → A},{A → B → E} respectively. In the case of p7:, the algorithm does not terminate, since dataset D2 is further connected to D3, so further iterations are performed.

Lets say two iterations ($Itr_{1,2}$) were carried out for previous datasets. Hereon, we will use Itr_i for iteration number i.

At Itr_3, D2 becomes $curDataset$ = D2 and $sources$ set is initialized with {A, D} (i.e. common nodes of D1 and D2). During Itr_3, D3 becomes $nextDataset$ = D3. Common nodes (i.e., {B}) between D2 and D3 are added to $targets$. At the end of Itr_3, when {A, D} × {B} is checked against D2, we get P' {A → B} against D2.

At Itr_4, D3 becomes $curDataset$ = D3 and {B} is added to $sources$. During Itr_4, D1 becomes $nextDataset$ = D1. Common nodes (i.e. C) between D3 and D1 are added to the $targets$. At the end of Itr_4, when {B} × {C} is checked against D3, we get P' {B → C} against D3.

At Itr_5, D1 becomes $curDataset$ = D1 and {C} is added to $sources$. During Itr_5, D2 becomes $nextDataset$ = D2. Common nodes {A, D} between D1 and D2 are added to $targets$. At the end of Itr_5, when {C} × {A, D} is checked against D1, we get P' {C → D} against D1.

At Itr_6, D2 becomes $curDataset$ = D2 and {A, D} are added to $sources$. During Itr_6, Algorithm 6.3 finds no further dataset to explore (see p7: in Figure 6.5). At this stage the $targets$ will only be one (i.e., n_{target} = E). At the end of Itr_6, when {A, D} × {E} is checked against D2, we get P' {D → E},{A → B → E} against D2.

When the computation is completed for p7: (Figure 6.5), the set {P'} contains all the direct and indirect paths computed by Algorithm 6.3. Now, the mergePaths (see Algorithm 6.4) component takes the set of partial paths {P'} and produces the complete paths P.

A complete list of paths shown in Figure 6.2b is generated when datasets paths (p1:–p8:) are completed with the above-mentioned procedures.

6.3.4 Path Merger

When the QPPDs Algorithm 6.3 has finished its task, we get a set of partial paths P' against each dataset. This set is given to Algorithm 6.4, which assembles all the paths in such a way that we get a list of complete paths P. To explain the path merger algorithm, let's take our running example. We step through the two cases of direct paths and indirect paths for a given query where n_{source} = F and n_{target} = E.

CASE-1 DIRECT PATHS: We explained earlier that the direct paths are calculated when Algorithm 6.3 comes into contact with p1 as shown in Figure 6.5. In this case, we get two paths P' {F → K → A},{F → G → A} against D1, and also two paths P' {A → B → E},{D → E} against D2. Note that Algorithm 6.4 iteratively performs all steps, where it can go back-and-forth during the iteration. At Line 2 of Algorithm 6.4, it checks if any path in D1 either starts with n_{source} = F or ends with n_{target} = E. If true, it will be stored in P. In the D1 case there is no such path that satisfies the previous condition. So all paths P' of D1 are checked against the next dataset paths P'. At Line 4 of Algorithm 6.4, it checks if any path P' from D1 starting with n_{source} = F and ending with any node that is equal to the starting node of any path P'

Algorithm 6.4: Path Merger Algorithm.

```
   input : [𝒟]_p′;                                   /* P′ against relevant datasets */
   output: P ;                                       /* Set of distributed paths  */
1  for (𝒟_i)_{i=1}^n ≠ ∅ do
2      if [D_i]_p′.prefix = n_source ∧ [D_i]_p′.postfix = n_target then
3          P ← p′;
4      else if
         [D_i]_p′.prefix = n_source ∧ [D_i]_p′.postfix = [D_{i+1}]_p′.prefix ∧ [D_{i+1}]_p′.postfix = n_target
         then
5          P ← p′ ;                                                  /* store path */
6      else if [D_i]_p′.postfix! = n_target ∧ [D_{i+1}]_p′.prefix = [D_i]_p′.postfix then
7          [D_{i+1}]_p′ ← [D_i]_p′.concat([D_{i+1}]_p′)
8  end
9  return P ;                                                  /* return solution */
```

exists in D2. For p1, in this case, the condition becomes true and we get complete paths in P $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow E\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow E\}$. The path P′ $\{D \rightarrow E\}$ is discarded, as it does not satisfy the applied condition at Line 4 of Algorithm 6.4.

CASE-2 INDIRECT PATHS: In the case of indirect paths, when Algorithm 6.3 has computed datasets connectivity starting from p2 to p8 shown in Figure 6.5, we get the following paths P′ $\{F \rightarrow K \rightarrow A\}, \{F \rightarrow G \rightarrow A\}, \{C \rightarrow D\}$ against D1, paths P′ $\{A \rightarrow B \rightarrow E\}, \{D \rightarrow E\}, \{A \rightarrow B\}$ against D2, and also one path P′ $\{B \rightarrow C\}$ against D3. At Line 4 of Algorithm 6.4, we get two complete paths P $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow E\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow E\}$. However, Algorithm 6.4 does not terminate at this stage. At Line 6, Algorithm 6.4 concatenates the path from D1 and D2, i.e. $\{F \rightarrow K \rightarrow A \rightarrow B\}, \{F \rightarrow G \rightarrow A \rightarrow B\}$. In the next iteration when D1 is checked against D3, we get $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow C\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow C\}$ at Line 6 of Algorithm 6.4. The next iteration at Line 6 of Algorithm 6.4 generates $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow C \rightarrow D\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow C \rightarrow D\}$. In the next iteration, when Line 4 of Algorithm 6.4 checks the condition, we get complete paths P $\{F \rightarrow K \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow E\}, \{F \rightarrow G \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow E\}$ and Algorithm 6.4 is terminated.

6.4 EVALUATION

In this section, we present the evaluation and results of the QPPDs approach. We first explain the evaluation setup, followed by the evaluation results and discussion.

6.4.1 Experimental Setup

Since, to the best of our knowledge, there exists no publicly available benchmark to test distributed property path retrieval systems on top of distributed RDF datasets, we had to create a benchmark by ourselves. Now we explain the datasets and path queries used in our evaluation.

Datasets

In our evaluation, we used 8 datasets – Disease, hpoClass, doClass, phenotype, Protein, Variant, Gene, and pantherClass – from the life-sciences domain with a combined total of 7.26 millions of triples. The "Disease" dataset contains information about the disease associated with at least one gene. The "Human Phenotype Ontology" (hpoClass) and "Disease Ontology" (doClass) both individually provide classifications of the genes. The dataset "Phenotype" contains the cross-referenced IDs extracted from HPO. The dataset "Protein" contains the UniProt IDs encoded by genes. "Variant" is the dataset that contains the variants associated with diseases. The "Gene" dataset contains gene information associated to diseases. The dataset "PantherClass" classifies the genes' attributes according to molecular functions.

We chose these datasets because they are interconnected and contain resources that are relevant to each other. Some of the high-level statistics of these datasets are shown in Table 6.1. All of the datasets used in our evaluation are publicly available from disgenet providers[1].

SETTING: We loaded each dataset into different Fuseki server instances, with our baseline algorithm already integrated to each Fuseki server to calculate and find the paths for a given query request.

Table 6.1: Disgenet Datasets statistics.

Query	Triples	Subject	Predicates	Objects
Disease	738626	60130	12	489756
doClass	101	21	11	63
Gene	1056346	119522	12	834502
hpoClass	253	36	11	151
pantherClass	272	40	9	123
Phenotype	83292	8441	8	66249
Protein	160537	14635	8	117034
Variant	5225996	708405	16	3628674

Path Queries

We wanted to test our approach on the most complex SPARQL1.1 property path[2] queries, where we did not specify any *regex* expression, but supporting arbitrary path length based on the use of wildcards (i.e., $(:|!\ :)*$).

We chose a total of 12 path queries for benchmarking, where each path query contains the source and target resources (see Table B.1 in Appendix B) from selected benchmark datasets. The total number of possible paths P, the number of hops or nodes in the given path (along with max., min., avg., and std.), and the number of datasets (along with max., min., avg., and std.) involved in P are given in Table 6.2. While constructing the path queries, we considered carefully that paths between source and target must have multiple datasets involved. We fixed the query time-out to 90 seconds, meaning that if a query is not executed within this time limit, it is considered a failure.

1 http://rdf.disgenet.org/download/v5.0.0/
2 https://www.w3.org/TR/sparql11-property-paths/

Table 6.2: Various characteristics of the benchmark path queries.

Query	Path	Hops				Datasets			
	Total	max	min	avg	std	max	min	avg	std
1	22	9	1	5	2.82	4	1	2.25	1.5
2	257	37	3	20	10.37	3	3	3	0
3	1	3	3	1	0	3	3	2	0
4	1	1	1	1	0	1	1	1	0
5	1	1	1	1	0	1	1	1	0
6	1	3	3	1	1	3	2	2.5	0.5
7	1	2	2	1	0	3	1	2.5	0.68
8	1	2	2	1	0	3	2	2.5	0.5
9	45	1	10	5.5	2.87	3	1	2.5	0.5
10	9	2	8	5	2.23	3	3	2	0
11	5	4	4	5	0	4	4	2	0
12	5	4	4	5	0	4	4	2	0

Hardware and Implementation Setup

We conducted path experiments on a local setup (i.e., local network) to maintain the network cost as low as possible. The 8 datasets used in our benchmark were loaded into Fuseki server version (1.3.0 2015-07-25T17). We used a cluster of 8 machines (Ubuntu OS) with 2.9GHzx8 Intel Core i7 processors, 4GB of RAM, and 250GB of storage capacity to run 8 Fuseki servers. The QPPDs engine is implemented in Java 1.8, using the Jena API. To run QPPDs, we used a MacBook Pro (Mojave OS) with 2.6GHz Intel Core i5 processor, 16GB of RAM, and 500GB of storage capacity. The code and all the configurations are available on GitHub.[3]

Metrics

The metrics with which these queries were evaluated are: (i) index generation time (ii) index compression ratio[4], i.e, the index size to dataset size ratio, (iii) the number of sources selected for each query, (iv) the source selection time, (v) the number of hops in the paths, (vi) the time taken to retrieve k paths. The system also provides a SPARQL endpoint that contains information about the retrieved paths, following the schema described in Appendix C. On the GitHub page, we provide sample queries to check the different metrics.

Baseline

The focus of this chapter is to introduce a federation-based approach to path finding in distributed RDF datasets. To do so and considering the overhead involved in federation, we presented above several algorithms that include a number of features aimed at reducing the response time for individual queries. We therefore here assess the contributions of those features by comparing response times with and without those features, starting from a baseline corresponding to an implementation of QPPDs without the following optimisation steps:

1. **Nodes Connectivity:** As mentioned before, the QPPDs approach only searches for complete paths if the intermediate nodes have both child and parent nodes. This is because a node without children or parent will terminate the current path computation. Therefore,

3 https://github.com/InsightGalway/Path-Federation
4 Index ratio: indexSize/datasetSize

before doing any processing on the given node, the QPPDs approach first checks for its child and parent node. In the baseline approach, we are not applying this filter until the algorithm itself discovers that the path is complete and terminates the processing for the current path search.

2. **Streaming approach:** The QPPDs approach works in batches of paths (as explained in section 6.3.3), i.e, a streaming approach for path computation. The baseline algorithm does not work in batches.

3. **Requests Grouping:** Our distributed path computation algorithm sends multiple path requests to the underlying datasource, i.e., SPARQL endpoints. The QPPDs approach has this feature to combine multiple path requests into a single composite request. However, in the baseline algorithm, only one path request is sent to the endpoint at a given time.

4. **Same Source-Target Filtering:** It is also possible that for a given path query the source and target nodes are exactly the same. The QPPDs approach filters out such requests before sending them to the remote SPARQL endpoints, hence the number of requests is reduced.

The impact of these missing features on the performance of the baseline algorithm is discussed in the next section. While comparing with other path querying approaches would be valuable, at this point, that those are applicable to different contexts, and often unavailable, prevents us from being able to conduct a fair comparison.

6.4.2 Results

Index Generation Time

QPPDs and the baseline approach both use exactly the same index of the given RDF datasets. Even though it is a one time process (assuming that no dataset updates), the index should be generated in a reasonable amount of time. Figure 6.6 shows the index generation time for each of the benchmark datasets. We can clearly see that the index generation time is dependent upon the size of the underlying dataset. The maximum time to compute the index for the largest dataset, i.e Variant, is only 136.6 seconds. The overall time to compute the indexes for the complete benchmark datasets is less than 6 minutes.

Index Size

The index should be small in size for fast lookups at runtime. The contribution of each of the benchmark datasets to the size of the index is shown in Table 6.3. We can clearly see that the indexes generated by our approach are very small in size. The cumulative size of the QPPDs index generated for all 8 datasets is only 6.1MB.

source selection

As mentioned before, each of the benchmark path queries contains the source and target nodes, distributed in multiple datasets. Thus, it is possible that a given source or target node is found in more than a single dataset. We measure source selection results in terms of: (1) the number of datasets which contain the source and target nodes of a given path query (see Table 6.4), (2) the time required to identify the source and target datasets for the given path query (see Figure 6.7).

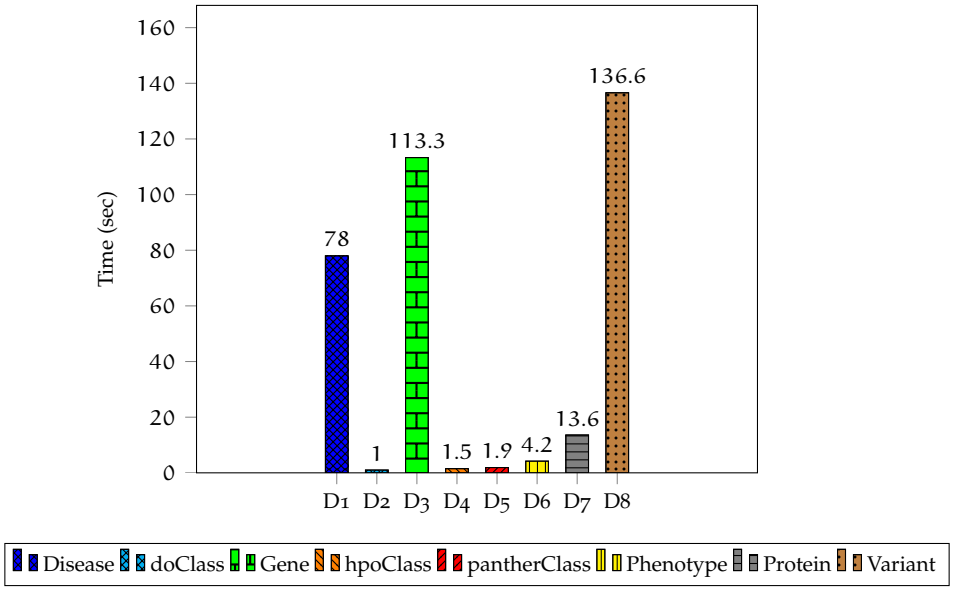


Figure 6.6: Index Generation Time.

Table 6.3: Index size for each dataset.

Dataset	Index Size (MB)	Dataset Size (MB)
Disease	0.3	116
doClass	0.004	0.014
Gene	3	170
hpoClass	0.013	0.038
pantherClass	0.008	0.047
Phenotype	0.015	12.6
Protein	2	22
Variant	1.1	995

Please note that the source selection of both baseline and QPPDs approaches is exactly the same, thus these results are the same for both approaches.

Table 6.4: Number of datasets selected for the source and target nodes of path queries.

Query	Source Dataset	Target Dataset
Q1	2	3
Q2	1	2
Q3	1	2
Q4	2	3
Q5	1	2
Q6	1	2
Q7	1	2
Q8	1	2
Q9	1	2
Q10	2	1
Q11	1	3
Q12	1	3

From the results given in Table 6.4 and Figure 6.7, we can see that the QPPDs approach is able to quickly filter out (requiring milliseconds) the irrelevant sources.

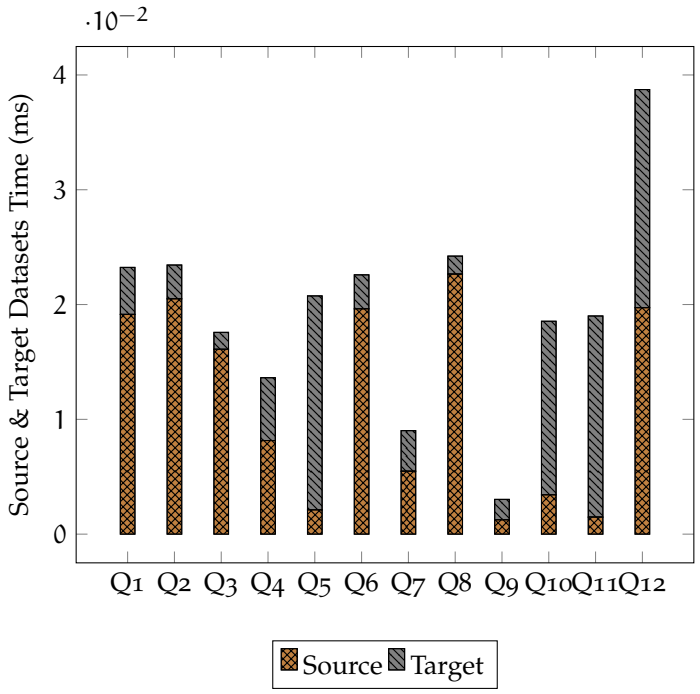


Figure 6.7: Time required to select the source and target datasets of benchmark path queries.

Query Runtime Performance

Figure 6.8 shows the query execution time for all 12 queries used in our evaluation. We observed that in two-third of the cases – for queries **Q1, Q2, Q3, Q4, Q6, Q7, Q9, and Q10** – QPPDs outperformed the baseline algorithm by almost one order of magnitude. It is because, for all these queries, the possible paths between the connected datasets are very high compared to other queries. Thus, these queries generate more endpoint requests.

The QPPDs, with its salient features (in particular checking for the child and parents of nodes beforehand), generates fewer requests as compared to the baseline approach. For the other queries **Q5, Q8, Q11 and Q12** the baseline approach performs better as compared to QPPDs. This is because these queries are simple and do not need a lot of time to be executed compared to other queries. QPPDs approach spends extra time for checking the child and parents of nodes involved in the final path, creating some overhead. We can conclude that the QPPDs approach generally performs better for complex path queries.

SCALABILITY OVER NUMBER OF SITES AND DATA SIZE: We noticed that the datasets involved in the results obtained from queries **Q1, Q2, Q3, Q4, Q6, Q7, Q9, and Q10**, are big in size (see Table 6.2) and have more connected nodes between them. This shows that the full QPPDs approach improves response time, as compared to the baseline, in the following situations for a given query:

- if the required path is distributed across more sites (datasets)
- if datasets are large in size

It is important to note that the comparison shown in Figure 6.8 is when a single path request is sent to the underlying triplestores, i.e, the *request grouping* feature of QPPDs is not activated.

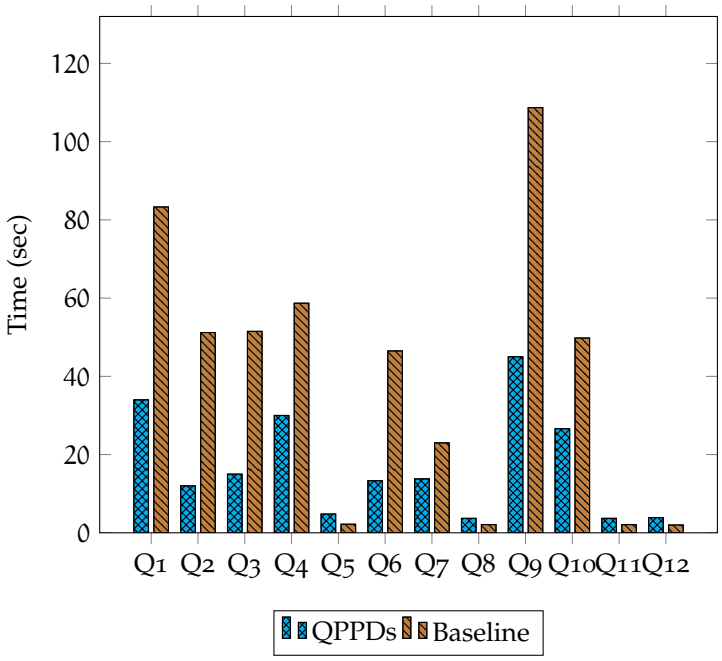


Figure 6.8: Query runtime performances of the QPPDs and baseline approaches when *request grouping* feature is not activated in QPPDs.

In the next section, we will see that the performance of QPPDs further improves with this optimization parameter.

Effects of QPPDs Optimization parameters

Now we show how much the query performance is affected either by enabling each of the QPPDs optimization parameters.

REQUESTS GROUPING: Figure 6.9 shows the query runtime performance by using a single request per query (QPPDs-1), grouping two requests per query (QPPDs-2) and grouping three requests per query (QPPDs-3). The overall results show that the performance is improved with the grouping of results.

In most of the queries the performance is further improved when we increase the grouping size of requests. However, for QPPDs-3, the queries **Q9, Q10** resulted in a time-out (90 sec). This leads to the question of whether the performance will be further improved if we further increase the grouping size. We tested with QPPDs-4 (i.e, grouped 4 requests per query) and noticed that the performance significantly dropped compared to QPPDs-3. We investigated the reason and noticed that the SPARQL endpoints hosting the underlying datasets were not able to handle more complex requests and hence started giving timeouts. Thus, the performance of QPPDs is also strongly dependent on the query processing capabilities of the underlying triplestores.

We believe that if efficient path finding algorithms on remote endpoints are deployed, the performance of QPPDs can further be enhanced.

SAME SOURCE-TARGET FILTERING: Figure 6.10 shows the effect of checking if the source and target represent the same node and hence skipping the path calculation process. Overall,

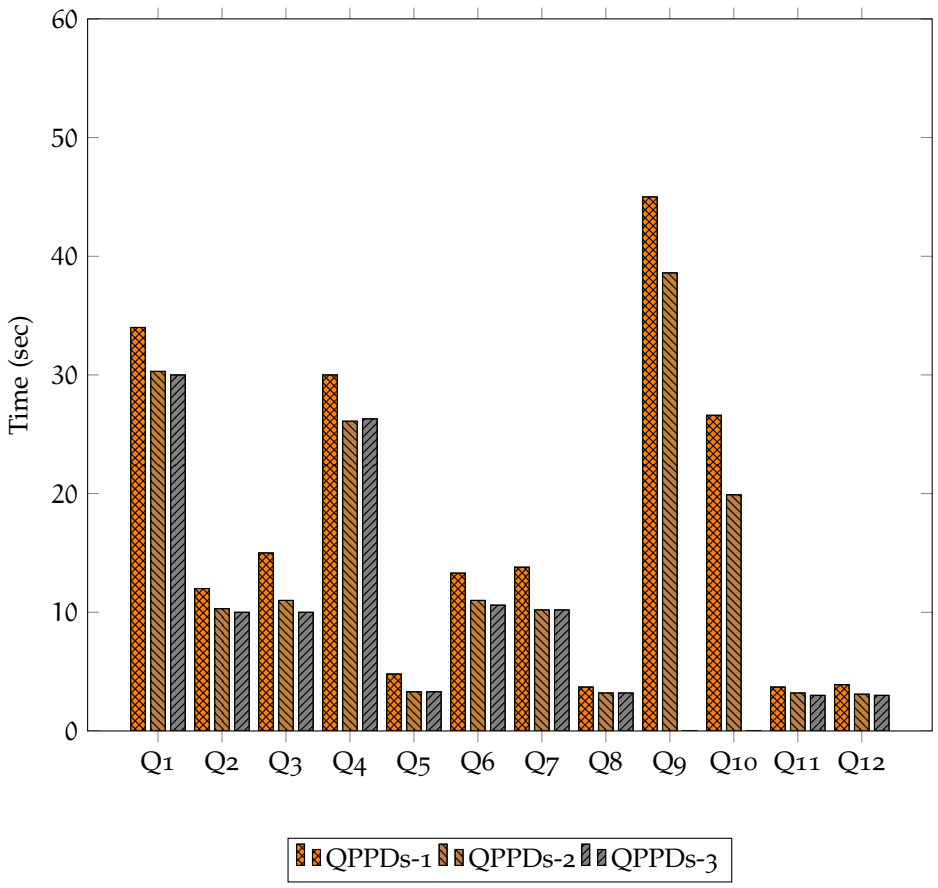


Figure 6.9: Effect of *request grouping* feature.

enabling this feature improved the query runtime performance for all of the 12 benchmark queries. On average, the response time is improved by 19% by enabling this feature.

NODE HAS CHILD: Figure 6.11 shows the effect of checking if a node has missing child nodes and if the current path is broken, i.e, it cannot reach the desired target node. Enabling this feature has resulted in improving the runtime performance for 9 out of 12 queries. For queries Q8, Q11, Q12 the performance is degraded. The reason for this is that the datasets involved in each query have more parent to child relations. However, the connected relations are not involved in the construction of the path. Hence, processing these relations takes more time compared to disabling this feature. On average, the runtime performance is improved by 22% by enabling this feature.

NODE HAS PARENT: Figure 6.12 shows the effect of checking if a node has a missing parent node and the current path is broken, i.e, it cannot reach the desired target node. Enabling this feature resulted in improving the runtime performance for 5 out of 12 queries. For queries Q3, Q5, Q7, Q8, Q10 and Q11 the performance is degraded. The reason for this is that the targeted datasets involved in query processing have more nodes that have parent nodes, and calculating these relations by enabling this feature takes more time. On average, the response time is improved by 17% by enabling this feature.

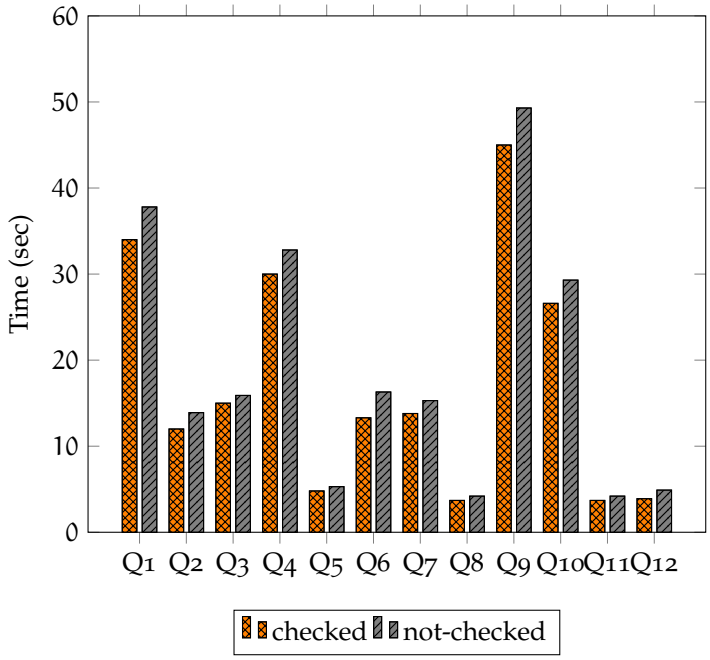


Figure 6.10: Effect of filtering same source and target nodes.

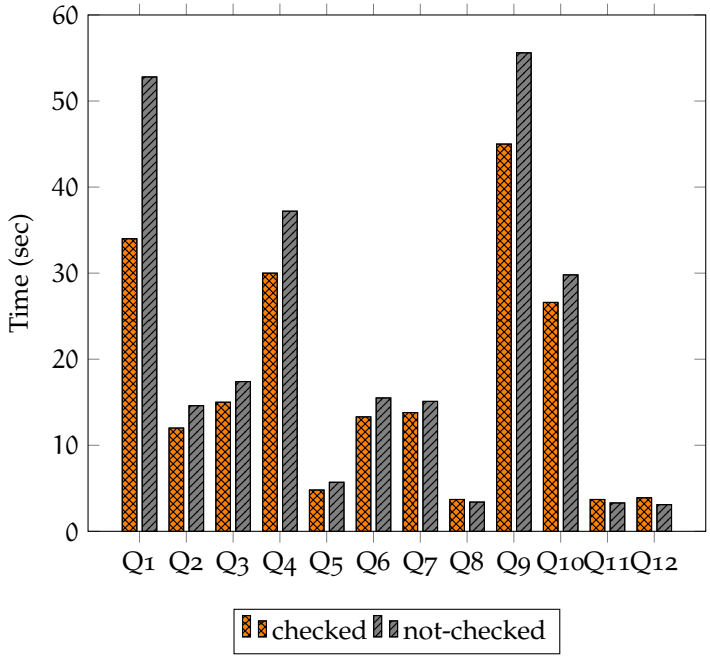


Figure 6.11: Effect of checking if a given node has child node.

Results Completeness:

To verify the QPPDs result completeness, we merged these 8 datasets into a single graph and tested against the centralised approach presented in chapter 4. Table 6.5 shows the comparison between the QPPDs, TopK, and baseline approaches in terms of *total number of paths* retrieved, maximum and minimum length (number of hops) of these paths. We noticed that for all queries the returned results are the same in the three approaches, except for one query. For query **Q2** the baseline algorithm returned fewer paths (i.e., 81) and with a maximum path hops of 15.

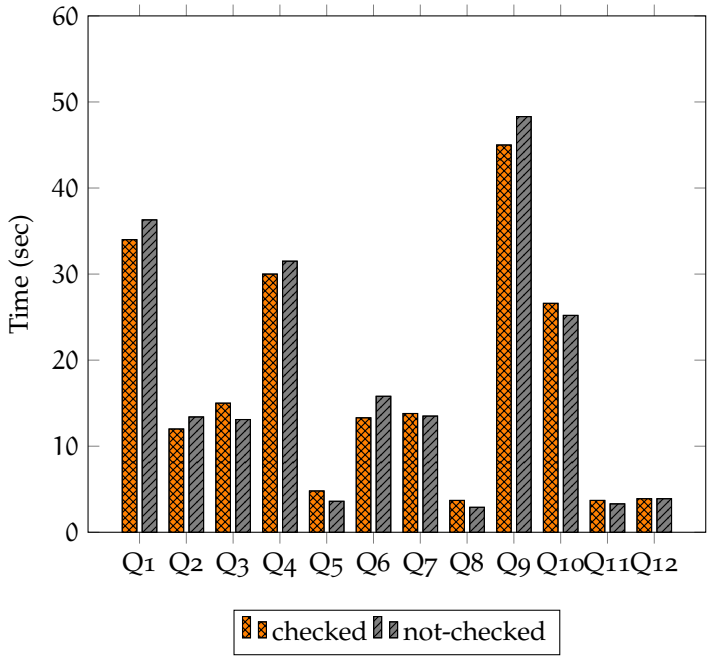


Figure 6.12: Effect of checking if a given node has parent nodes.

QPPDs however returned 257 paths with maximum path hops of 37. This is because the baseline algorithm sent more parallel requests to some of the remote endpoints, exceeding the endpoint's limit, which started to give the exception "*cannot assign requested address*".

Table 6.5: Comparison of the paths retrieved by the baseline approach and our QPPDs approach.

Query	QPPDs		Baseline		TopK	
	tot-paths	max-hops	tot-paths	max-hops	tot-paths	max-hops
1	22	9	22	9	22	9
2	**257**	37	**81**	15	**257**	37
3	1	3	1	3	1	3
4	1	1	1	1	1	1
5	1	1	1	1	1	1
6	1	3	1	3	1	3
7	1	1	1	1	1	1
8	1	2	1	2	1	2
9	45	10	45	10	45	10
10	9	8	9	8	9	8
11	5	4	5	4	5	4
12	5	4	5	4	5	4

6.5 CONCLUSION

In this chapter, we proposed QPPDs, a path traversal approach that federates path queries across multiple SPARQL endpoints. As the size of Linked Data is increasing and more and more datasets are made publicly available through different endpoints on Linked Open Data cloud 2.3, the approaches we proposed previously in Chapters 4 and 5 become insufficient. The concept behind the QPPDs approach is tackling those challenges that were described for centralised envi-

ronments in Chapter 4 and for homogeneous approaches that work in controlled environments in Chapter 5.

Different database systems, not necessarily focusing on RDF, have already started supporting the enumeration of paths beyond the current SPARQL 1.1 Property Path specification. These include the Stardog [Sir] system and the systems mentioned in Table 3.1 (see Column "Output Paths") of Chapter 3. The current implementation we provided here is tested on Fuseki server [Jena] running with a common Breadth First Search (BFS) algorithm. QPPDs could however work with such systems that provide path enumeration "off-the-shelf" and could be easily adapted for systems relying on different graph models (e.g., weighted, unweighted, property, etc.).

Another salient feature, which is not available in other state-of-the-art approaches, is that QPPDs provides a dataset, exposed as SPARQL endpoint, of paths that contains statistical and provenance information for each path.

We can notice that QPPDs has advantages over FedS. This is due the concept and implementation of QPPDs that can work in a heterogeneous environment. On the other hand, FedS was implemented as a P2P approach that was able to work only in a controlled environment. With all its advantages over FedS, QPPDs, however, depends on a precomputed index, and due to this dependency, the paths and results completeness are not assured if the index is not up-to-date according to the underlying data on a remote dataset. Consequently, in the next chapter 7, we propose an index-free distributed path query engine called DpcLD that has the ability of querying the paths even if the data is updated in remote endpoints. Furthermore, our DpcLD is more robust and efficient compared to FedS and QPPDs, and is tested form a large amount of data.

7 CACHE ASSISTED INDEX-FREE DISTRIBUTED PATH QUERY APPROACH

This chapter presents our third approach towards distributed path query processing. In previous chapters (5 and 6), we proposed FedS that works in a P2P environment and QPPDs a path query federation engine that has the capability of querying distributed paths in heterogeneous datasets. Here, we propose a path query engine – called DpcLD (Distributed Path Computation over Linked Data) — where users can execute path queries and find paths distributed across multiple, connected graphs exposed as SPARQL endpoints. DpcLD is an index-free, cache-based query engine that communicates with a shared algorithm running on each remote endpoint, and computes distributed paths. In this chapter, we highlight the way in which this approach exploits and aggregates partial paths, within a distributed environment, to produce complete results. We perform extensive experiments to demonstrate the performance of our approach on two datasets: One representing 10 million triples from the DBpedia SPARQL benchmark, and another full benchmark dataset corresponding to 124 million triples. We also perform a scalability test of our approach using real-world genomics datasets distributed across multiple endpoints. We compare our distributed approach with other distributed and centralized pathfinding approaches, showing that it outperforms other distributed approaches by orders of magnitude, and provides a good trade-off for cases when the data cannot be centralised.

The structure of this chapter is organised as follows: Section 7.1 highlights the motivations. In Section 7.2 we explain the architecture of the system and our approach with a running example to make the core components easy to understand. In section 7.3, we provide an extensive evaluation both for synthetic and real-world data, and provide comparisons with state-of-the-art approaches (i.e., distributed and centralized). Finally, Section 7.4 summarizes and concludes this chapter.

7.1 MOTIVATION

Most approaches to distributed path queries, including FedS presented in Chapter 5, work in homogeneous and restricted environments. Some of them, which include our approach QPPDs proposed in Chapter 6, require a precomputed up-to-date index. Hence, path completeness is only assured if the index is up-to-date according to the current status of the underlying distributed RDF datasets. Furthermore, some of these approaches, e.g., [HP17], are unable to perform lookups for non-dereferenceable URIs [UHP+15]. Hence, they may retrieve empty or incomplete results. Note that our previous work [VSN+18] shows that around 43% of the URIs of more than 660K RDF datasets from LODStats [ELM+16] and LODLaundromat [BRB+14] are non-dereferenceable.

In this chapter, we address existing challenges and limitations in the state-of-the-art approaches. Our implementation includes (i) DpcLD, a cache-assisted pathfinding engine, and (ii) a shared algorithm, distributed and running on remote endpoints, that communicates with the DpcLD engine. Moreover, our approach is index-free, efficient and does not require dereferenceable URIs, therefore, returning complete results even after the datasets have been updated.

7.2 DPCLD

DpcLD is a path query processing engine that computes K paths between two nodes from distributed RDF graphs while communicating with a shared algorithm running on remote endpoints (triple stores). The DpcLD architecture is summarised in Figure 7.1, which shows its core components, i.e., (i) Source Selection: performs source selection and selects a relevant datasource to start the traversal from, (ii) Federated Path Computation: once a candidate datasource is selected, DpcLD starts path traversal, interacts with the (iii) Cache, and dispatches the queries to remote triple stores where it is required, and finally (iv) Paths RDFizer: breaks down (triplize) all retrieved paths (i.e., complete or partial paths) and stores into a temporary graph where a Bidirectional-BFS pathfinding algorithm computes the K paths.

In summary, posing a path query, the DpcLD engine delegates the requests to the data sources. The instances of the shared algorithm 7.2, running on remote endpoints, compute the paths (full or partial) against each query request and return the answers back to engine. The DpcLD engine, on receiving these answers, intelligently computes and generates the complete paths and finally the results are presented to the user.

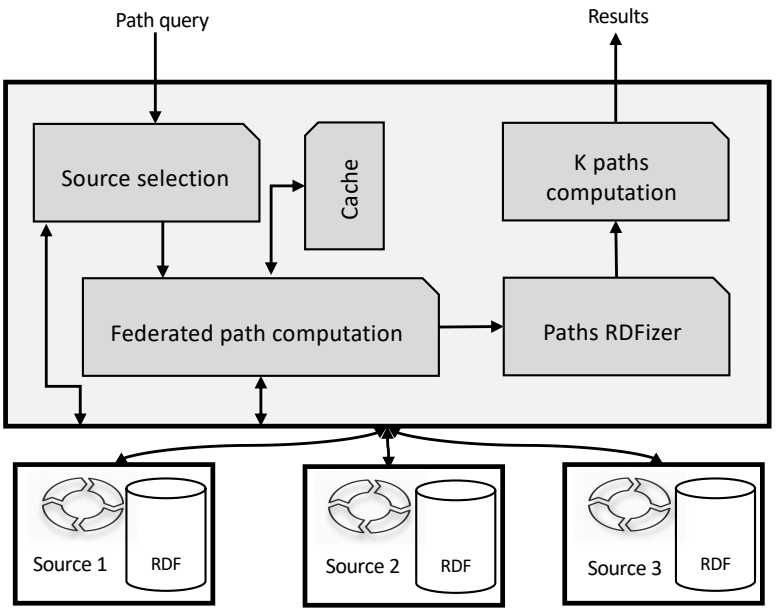
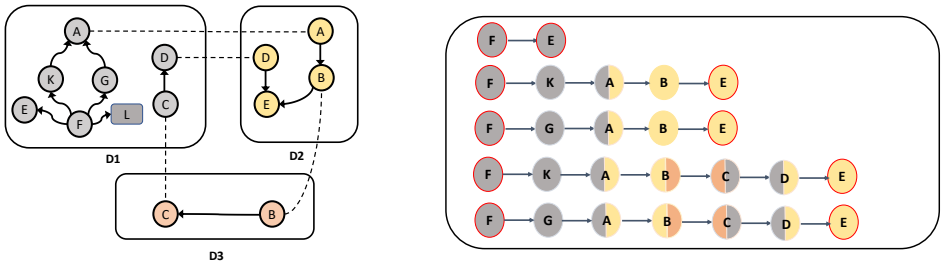


Figure 7.1: DpcLD's high level architecture.

7.2.1 Running example

Figure 7.2a shows three fictional datasets distributed across the network and their connectivity. Suppose a user poses a path query PQ to DpcLD, through which s/he wants to find the K paths between a source F and target E, with $K = 5$. K represents the number of paths desired. As shown in Figure 7.2a, dataset D1 contains only a single path between F and E. However, dataset D1 is connected to other datasets (i.e., D2, D3) via some links. To fulfil the K requirement, the DpcLD engine exploits these links and finds partial paths from remote datasets and rearranges these to produce complete paths. Figure 7.2b depicts the generated paths that involve different datasets.



(a) Sample datasets and their connectivity.

(b) Sample paths from the example.

Figure 7.2: Running example: Datasets and paths between node F and node E.

7.2.2 Data Model and Query

In this chapter, we again consider RDF directed labeled graphs (see 2.1.2). Each graph is loaded into a triple store, providing a SPARQL endpoint over the data accessible through the network. We have defined RDF graph and different terminologies in Section 3 of Chapter 2. We also defined paths and their characteristics in Section 2.1.1 of Chapter 2.

To compute K paths between two nodes, DpcLD uses the query template (see Listing 16) that we proposed in chapter 4. The query shown in Listing 16 finds and enumerates 5 paths (see Figure 7.2b) between source :F and target :E

```
PREFIX : <http://insight-centre.org/sample/>
PREFIX ppfj: <java:org.centre.insight.property.path.>

SELECT * WHERE { ?path ppfj:topk (<:F>  <:E>  5) . }
```

Listing 16: SPARQL query to find the common nodes between datasets

Our defined path query PQ is denoted by $PQ = (s_i, t_i, K)$, where $s_i \in I$ is the source node, $t_i \in I$ is the target node, and $K \in \mathbb{Z}^+$ is the number of paths to be retrieved. The result set of such a query PQ on a graph G is the set $[\![PQ]\!]_G$ of up to K paths connecting s_i to t_i in G. For the purpose of describing how DpcLD works, we additionally rely on the notion of partial path:

Definition 9 *Partial Path: A Partial Path* $P'(n_1, n_k)$ *is a path* $P(n_1, n_i)$ *which does not include* n_k. *In other words, it is a path starting at the source node, but which has not reached the target node. A path* $P(n_1, n_k)$ *is therefore referred to in this chapter as a complete path.*

DpcLD relies on computing partial paths in individual graphs, and reconnecting them in order to form complete paths across the overall federated graph. We denote as $[\![PQ_G$ the set of partial paths $P'(s_i, t_i)$ resulting from the path query PQ in the graph G.

Formally, considering a set of federated, local graphs $\mathcal{D}$, the objective of DpcLD is to compute the set of paths $[\![PQ]\!]_{\mathcal{G}}$ where $\mathcal{G}$ represents the union of the graphs $G_i \in \mathcal{D}$. To simplify, we will also refer to this result set as $[\![PQ]\!]_{\mathcal{D}}$, substituting the set of graphs $\mathcal{D}$ with its union $\mathcal{G}$. DpcLD relies on a process to achieve this which minimises repeated computations and total network overhead.

7.2.3 DpcLD: Algorithm

To evaluate a path query PQ over distributed datasets $G_i \in \mathcal{D}$, the DpcLD engine not only finds the path set $[\![PQ]\!]_{G_i}$ within each graph, but also partial paths $[\![PQ_{G_i}$. Before we explain the DpcLD system in details, it is important to understand what type of cases the system has to cope with for a given path query $PQ = (s, t, K)$:

CASE 1: if $s \in G_i \wedge t \in G_i$, i.e. both the source and target nodes are within the local graph G_i, and the number of local paths satisfy K, then these will be directly pushed to the solution list and the algorithm will be terminated

CASE 2: if $s \in G_i \wedge t \in G_i$, but the local paths do not satisfy the required K, then the algorithm will query the other graphs in $\mathcal{D}$ for (complete and partial paths).

CASE 3: if $s \in G_i \wedge t \notin G_i$, i.e. the source node is in the local graph, but not the target node, then the algorithm will query the other graphs in $\mathcal{D}$ for partial paths.

The pseudocode in Algorithm 7.1, shows how a path query $PQ = (s, t, K)$ is processed over the set of graphs $\mathcal{D}$. The steps performed are described in the following subsections.

Source Selection

DpcLD performs a partially index-free source selection of relevant graphs (triple stores, see Definition 10), where only the address URI of the endpoint for each graph is stored. The source selection of relevant data sources is performed by selecting the dataset in which the source node s has more outgoing links. For instance, in our running example, we assume a set of graphs $\mathcal{D} = G_1, \ldots, G_n$, where each graph G_i is a dataset accessible through a SPARQL endpoint. When a user poses a path query $PQ = (:F, :E, K)$, the DpcLD engine probes the relevant datastore in all datasets (Lines 3-9 of Algorithm 7.1). Figure 7.2a shows that only dataset D1 contains the source node :F. Therefore DpcLD, in this case, considers D1 as a local graph and starts computation on this dataset. We formalise the relevant data sources for s in the following definition:

Definition 10 *Source Datasets: A graph* $G_i = (V_i, E_i) \in \mathcal{D}$ *is a source dataset in the context of a path query* $PQ(s, t, K)$ *if* $s \in V_i$ *and there is at least one triple in* E_i *with* s *as subject.*

Federated Path Computation

This component computes the paths, communicates with the cache, and dispatches requests where required. The cache here plays a vital role in terms of performance and reducing the

Algorithm 7.1: Pseudo-code of the algorithm implemented by DpcLD.

```
   Input  : A query Q = (s, t, K) over 𝒢
   Output : The answer set [[Q]]_𝒢
1  𝒟 = {D_1, ..., D_n};                                   /* index of data sources URIs */
2  D_l ← D_1;
3  max_oeg ← 0;
4  foreach D ∈ 𝒟 do
5  |  outgoing ← findOutGoingEdge(s, D);
6  |  if |outgoing| > |max_oeg| then
7  |  |  max_oeg ← outgoing;
8  |  |  D_l ← D
9  end
10 sol ← ∅;                                               /* solution (retrieved complete paths) */
11 q ← {T_node(s, null, null)};  /* initialise q as T_node which contains triple(tp) binary relations as transitive
     closure */
12 while q ≠ ∅ do                                        /* check until queue is empty */
13 |  tp ← q.poll()                                       /* take triple tp from current T_node */
14 |  for (tp_i)_{i=1}^n ∈ D_l do                          /* iterate over all (p;o) of s for current triple tp */
15 |  |  if (tp(o)_i = literal) ∨ (isVisted(tp(o)_i) = true) then   /* if o is literal or current triple
   |  |    tp_i | (s;p;o) is already visited then skip */
16 |  |  |  continue;
17 |  |  if (tp(o)_i = t) then                           /* if target node found */
18 |  |  |  P ← path(tp, tp(p;o)_i);                     /* append tp(p;o) to the tp as a complete path */
19 |  |  |  sol ← P;                                     /* solution is found */
20 |  |  |  if |sol| ⩾ K then
21 |  |  |  |  return sol;                              /* if K is satisfied then terminate the algorithm (i.e., case:1) */
22 |  |  |  end
23 |  |  |  CacheCheck(P, 𝒟);                           /* check the cache and perform necessary tasks */
24 |  |  else
25 |  |  |  P′ ← path(tp, tp(p;o)_i);                   /* append tp(p;o) to the tp as a partial path */
26 |  |  |  q ← q.add(tp_i);                            /* update queue */
27 |  |  |  CacheCheck(p′, 𝒟);                          /* check the cache and perform necessary tasks */
28 |  end
29 end
30 tempModel ← RDFizer(Cache.getPartialAndfullPaths);   /* when Case:1&2 are not satisfied and local computation is
     finished */
31 sol ← BidirectionalBFS(s, t, K, tempModel);          /* run a bidirectionalBFS algorithm on temp Graph and computes the K
     paths */
32 return sol;                                          /* return solutions */
33 Function CacheCheck(path, 𝒟)
34 |  foreach dataset D ∈ 𝒟 do
35 |  |  foreach (path_node)_{node=1}^n;                 /* iterate over each node of a path (P or P′) */
36 |  |  do
37 |  |  |  visited ← Cache.check(node, D);             /* check each node of path if visited against current dataset */
38 |  |  |  if visited = true then
39 |  |  |  |  continue;
40 |  |  |  else
41 |  |  |  |  p_r ← FedRequest(node, t, D);           /* obtain remote path through federated request */
42 |  |  |  |  PathStatus(p_r, D)
43 |  |  end
44 |  end
45 Function PathStatus(pathLst, D)
46 |  foreach path ∈ pathLst do
47 |  |  if path.startNode=s ∧ path.endNode=t then      /* check if current is a complete path P */
48 |  |  |  P ← path;
49 |  |  |  Cache.put(D, BrkPath_Into_Nodes(P), P)
50 |  |  else
51 |  |  |  P′ ← path;
52 |  |  |  Cache.put(D, BrkPath_Into_Nodes(P′), (P′))
53 |  end
54 Function BrkPath_Into_Nodes(path)                    /* break path into indvisual nodes */
55 |  nodes ← {∅};
56 |  foreach (path_node)_{node=1}^n;                    /* iterate over each node of a path (P or P′) */
57 |  do
58 |  |  nodes.Add(node);;                               /* each node from path is added to the visited nodes list */
59 |  end
60 |  return nodes;
```

number of HTTP requests. The DpcLD engine using cache avoids sending duplicate requests. We implemented the cache as a *key-value pair* store, where every dataset $D \in \mathcal{D}$ is a key and has multiple values (*visited nodes*, *full paths* P, *partial paths* P′) against each key.

In our running example, the traversal starts (Line 10 of Algorithm 7.1) and the queue q is initialized with an object T_{node} which at the beginning contains a triple with only a source node :F as subject. The object T_{node} holds paths for each node as a transitive closure, i.e., it contains

the *incoming edge, the current vertex,* and its *previous vertex (predecessor)*. For instance, given a path query PQ with a source node :F, the traversal starts from :F and iterates over each of its triples/successors T_i := (current vertex|s;p;o) (Line 14). The algorithm checks two conditions: (1) if any successor i.e., o is not a URI (i.e., a literal), and (2) if o is already visited for :F as a predecessor (to check for *cycles*). If any of these conditions is true, the particular iteration T_i will be skipped and the algorithm continues (Lines 15-16) to the next iteration. For example, a connected object :L to the source node (:F→L) is a literal, as depicted in Figure 7.2a. Thus, it will be skipped in the iteration $i = 1$.

Other than the previous two conditions explained above, the algorithm navigates along all the paths *(successors)* of :F (Lines 17-27) and checks the reachability (see Section 2.1.1 of Chapter 2) between source :F and target :E node.

In our running example, at iteration $i = 2$, the algorithm finds a complete path P (i.e.,:F→:E) within the local dataset D1, and the situation in this iteration represents the case 2. The algorithm does not terminate at iteration $i = 2$ but performs the following steps and updates the cache against the dataset D1:

- stores the path P (:F→:E) in the full path set.
- gets all the nodes (i.e., :F and :E) from the current path and put them in the *visited nodes* set against D1.
- *in parallel* checks the cache to see if these node are visited against other datasets. If the cache does not have these nodes visited for other datasets, it sends requests (i.e., *Q (:F,:E)*) to remote datasets (i.e., D2, D3, see Line 41).

Since the current path contains only two nodes, one request *PQ (:F,:E)* is dispatched to each remote datasets. It is notable that multiple requests will be dispatched in those cases where more than two nodes exist within a path. At current iteration $i = 2$, we can see that remote datasets did not return any paths, however, we still treat :F, :E as visited nodes against each dataset D2, D3 and store them as (*visited nodes*) in the cache. Table 7.1 shows the cache storage when iteration $i = 2$ is completed. The set of *visited nodes* is an important factor to minimise the number of remote requests because each element (i.e., visited node) stored in this set against individual datasets will not be counted for future requests, hence, reducing the network cost.

Table 7.1: Cache at $i = 2$.

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,E]	[(F→E)]	[]
D2	[F,E]	[]	[]
D3	[F,E]	[]	[]

Before we explain the next steps, we can see that only one path has been found and the solution still does not satisfy the K parameter (i.e., 5). Therefore, the algorithm continues to next iterations. When the algorithm encounters the iterations $i = 3$ and $i = 4$, then it has to cope with the situation that falls under the case 3, where only the source node :F does exist within the dataset D1. The algorithm (Line 25) finds two incomplete paths, at $i = 3$ (F→K→A) and at $i = 4$ (F→G→A) respectively, in local datasets D1. We denote such paths as *partial paths* P/ which do not reach target :E from source :F.

After iteration $i = 3$ is completed, the cache is updated and the partial path (F→K→A) along with the *visited nodes* are stored against key D1. In *parallel*, the algorithm constructs the path queries and delegates those to remote triple stores i.e., D2, D3. The way the queries are generated is discussed in the following subsection.

Query Generation

As explained earlier, the number of generated queries directly depends on the nodes in a path. For example, for the path (F→K→A), the possible queries could be; (i) *PQ(:F,:E)*, (ii) *PQ(:K,:E)*, and (iii) *PQ(:A,:E)*.

However any node that is already visited, against relevant datasets $D \in \mathcal{D}$, will not be considered in the query generation. Moreover, while constructing a query, where source and target are the same (e.g., *(:E,:E)*), the algorithm will not generate the corresponding query. These two checks improve the query performance significantly by reducing the number of unnecessary remote requests.

It is also important to note while generating the requests that each query PQ will always have the same target:E.

For path (F→K→A) at iteration $i = 3$, the algorithm checks in the cache (Line 37) which nodes already exist against relevant datasets in $\mathcal{D}$. We can see that :F, :E are there against all datasets. Therefore, the query *PQ(:F,:E)* will not be generated and only two queries, i.e., *PQ(:K,:E)* and *PQ(:A,:E)*, are constructed and dispatched to datasets D2 and D3 (not D1 since :K and :A are already visited in D1). After receiving the responses from remote datasets (i.e., D2 and D3), the path status (Line 42) is checked and the cache is updated, where a partial path (A→B→E) against D2 is returned and no path against D3. The same procedure, as for $i = 3$, is followed by Algorithm 7.1 at iteration $i = 4$ where, for the path (F→G→A), only one query *PQ(:G,:E)* is constructed and dispatched to D2 and D3. The queries for node :F, :K, :A are not generated since these nodes are already visited against all datasets. For query *Q(:G,:E)*, both D2 and D3 return no result. However, their *visited node* sets are updated with :G in the cache. After the completion of $i = 3$ and $i = 4$ respectively, the updated cache contains what is shown in Table 7.2.

Table 7.2: Cache at $i = 3$ and $i = 4$.

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,K,G,A,E]	[(F→E)]	[(F→K→A), (F→G→A)]
D2	[F,K,G,A,B,E]	[]	[(A→B→E)]
D3	[F,K,G,A,E]	[]	[]

In our running example, when the iteration $i = 4$ is finished, the local traversal for D1 is terminated because there is no more possible paths P or P′ from :F to its triples/successors $T_i := (\text{current vertex}|s;p;o)$ within D1.

Until now, the values *full path* or *partial path* in the cache included only direct paths, i.e., paths leading from D1 to D2 and D3. In the following definition, we explain the notion of direct path.

Definition 11 *Direct Paths: A direct path is a path where at most two datasets are involved. For instance,* D1 *or* D2 *contributed to calculate the first three paths shown in Figure 7.2b.*

However, in many cases the paths, could be *indirect paths*, as shown in Figure 7.2a where D1, D2 and D3 participate to complete the 4th and 5th paths in Figure 7.2b. We define indirect paths in the following definition:

Definition 12 *Indirect Paths: An indirect path is a path involving more than two datasets. For instance,* D1, D2 *and* D3 *contribute to calculate the 4th and 5th paths shown in Figure 7.2b.*

When local traversal is finished at dataset D1, however, we can get only 3 paths, (1, 2, and 3) (see Figure 7.2b), and these are the direct paths where D1 and D2 contributed. We still miss the desired $K = 5$ solutions. How the remaining 2 paths are calculated is explained in the following paragraph.

After the iteration $i = 4$ when local traversal has finished, the algorithm does not terminate but recursively starts checking the *difference between sets of visited nodes* $A\Delta B = \{x : [x \in A \text{ and } x \notin B] \text{ or } [x \in B \text{ and } x \notin A]\}$ from one dataset to every other datasets within the cache $|PC|$. If any non-overlapping node – visited against one dataset but not for others – is found in a particular dataset then it is also checked against other unexplored datasets. For instance, in Table 7.2, node :B is in the *visited nodes* list against D2, but not against D1 and D3. So, when node :B is checked against these remaining datasets, i.e., D1 and D3, we get updated values in cache as shown in Table 7.3.

Table 7.3: *Cache update when :B is checked against* D1 *and* D3.

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,K,G,A,B,E]	[(F→E)]	[(F→K→A), (F→G→A)]
D2	[F,K,G,A,B,E]	[]	[(A→B→E)]
D3	[F,K,G,A,B,E,C]	[]	[(B→C)]

In the previous step performed for Table 7.3, we can see, in dataset D3, that node :C in the *visited nodes* list is not yet checked against other datasets. When node :C is checked against datasets D1, D2, we get an updated list as shown in Table 7.4.

Table 7.4: *Cache update when :C is checked against* D1 *and* D2

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,K,G,A,B,E,C,D]	[(F→E)]	[(F→K→A), (F→G→A),(C→D)]
D2	[F,K,G,A,B,E,C]	[]	[(A→B→E)]
D3	[F,K,G,A,B,E,C]	[]	[(B→C)]

As shown in Table 7.4, when the cache is updated, the node :D would appear in the difference between dataset D1 and others. When :D is checked for paths against D2, D3, the cache is updated as shown in Table 7.5. When all of the *visited nodes* appear for all datasets, Algorithm 7.1 stops checking cache and is terminated.

It is important to note that *visited nodes* in cache are only those that contribute to complete P or partial P′ paths, and therefore include a subset of the graph only.

Table 7.5: *Cache update when :D is checked against* D2 *and* D3.

Keys	Values		
Dataset	Vist.Nodes	F.Paths	P.Paths
D1	[F,K,G,A,B,E,C,D]	[(F→E)]	[(F→K→A), (F→G→A),(C→D)]
D2	[F,K,G,A,B,E,C,D]	[]	[(A→B→E),(D→E)]
D3	[F,K,G,A,B,E,C,D]	[]	[(B→C)]

Paths RDFizer:

When the main algorithm 7.1 is terminated, all the paths are broken down into triples by our *path rdfizer algorithm*. This is a simple mechanism, since in a path every node is connected to other node through one hop, like a triple (i.e., $T := (s; p; o)$) and we break a path in such a way that every object o becomes the subject s for a triple with the next connected node as object. For example, a path *(F→K→A)*, when triplified, will generate two triples: (i) $F \xrightarrow{p_1} K$, and (ii) $K \xrightarrow{p_3} A$.

An excerpt shown in Listing 7.1 represents the RDFized (N-Triples) data for *partial paths p'* retrieved in Table 7.5.

```
<http://node-C> <http://property-p6> <http://node-D> .
<http://node-A> <http://property-p7> <http://node-B> .
<http://node-F> <http://property-p4> <http://node-G> .
<http://node-F> <http://property-p1> <http://node-K> .
<http://node-K> <http://property-p3> <http://node-A> .
<http://node-D> <http://property-p6> <http://node-E> .
<http://node-B> <http://property-p7> <http://node-E> .
<http://node-B> <http://property-p8> <http://node-C> .
<http://node-G> <http://property-p5> <http://node-A> .
```

Listing 7.1: N-Triples format of paths given in Table 7.4.

A local traversal algorithm (presented in chapter 4) is executed on this temporary graph G_{tmp} and all the *complete paths p* returned by this algorithm are added into the `solution` set. In the end, the query shown in Listing 16 returns the *answer set* $[\![Q]\!]_{\mathcal{D}}$ to the user with the $K = 5$ results.

7.2.4 Shared Algorithm

In the previous section, we explained that the DpcLD engine communicates with a shared algorithm. This algorithm (Algorithm 7.2) is distributed and deployed on remote SPARQL endpoints, to calculates the `paths`, either complete P or partial P*′*. It is implemented as a standard Breadth First Search (BFS) process with some modifications.

Lines 1-4 are the inputs for the algorithm. At Line 6 a `queue` q of type T_{node} is initialised with the start node s. The T_{node} object stores the triples in a chain such that the predecessor of each current triple can be accessible. At Line 6, the first time when Algorithm 7.2 is started, there will not be any predecessor for the source node s. At Line 8 the current triple is pulled from the `queue` q and at Lines 9-24 the algorithm traverses each vertex or the `object` o of that particular triple. At Line 10, the algorithm checks and continues to the next iteration if the `object` value of

Algorithm 7.2: Algorithm to find K paths locally between source and target datasets.

```
1  s← source ;                                                    /* source node */
2  t← target ;                                                    /* target node */
3  k← K_paths ;                                  /* search number of TopK paths */
4  D← G ;                                                         /* dataset */
5  sol← ∅ ;                                    /* solution (retrieved properties) */
6  q← {T_node(s, null, null)} ;        /* initialise q with Object T_node which can store
     current triple(tp) and all its predecessors */
7  while q ≠ ∅ do                                  /* check until queue is empty */
8      tp ← q.poll()                              /* take tp as current T_node object */
9      for (tp_i)_{i=1}^{n} do    /* iterate over each child for a specific triple object */
10         if (tp(o)_i = literal) ∨ (isVisted(tp(o)_i)) then          /* move next */
11             continue;
12         if (tp(o)_i = t) then
13             sol ← path(tp, tp(p; o)_i) ;      /* append tp(p; o) to the tp as a complete
                 path P  */
14         else
15             if (q.contains(tp_i) then    /* move next if queue q already contains this
                 triple  */
16                 continue;
17             sol ← path(tp, tp(p; o)_i) ; /* append tp(p; o) to the tp as a partial path
                 P/ */
18             q← q.add(tp_i)
19             if sol.size ⩾ K then
20                 return sol;
21             end
22     end
23 end
```

the current (s,p,o) triple is a *literal* or is already *visited*. At Line 13, if the leading *object* value is the actual target t, the algorithm adds it to the solution. At Line 15, the algorithm checks and continues to the next iteration if the *queue* already contains this triple in any of the triple's *predecessor* or *successor*. Line 15 is an extra check which allows only to store distinct paths and, therefore, restricts the unnecessary increase in the *queue* size. At Line 17 the calculated paths P/ are also stored into the solution list. At Line 19, if the size of the solution list, which contains both P and P/, becomes greater than K, Algorithm 7.2 terminates and returns the solution (i.e., complete P and partial P/ paths) towards the DpcLD engine.

7.3 EVALUATION

In this section, we explain the evaluation setup, the synthetic and real-world datasets used, the queries and the results of our experiments.

7.3.1 Experimental Setup

In this section, we discuss the datasets, the set of path queries over the selected datasets, the performance metrics we used in the evaluation, and the state-of-the-art path finding systems over Linked Data, we used in our evaluation.

Datasets

We used both synthetic and *real-world* datasets in our evaluation. The synthetic datasets[1] were selected from the ESWC-2016 shortest path finding challenge and real − world datasets were selected from biological data[2] provided by DisGeNET. We have already used the ESWC-2016 shortest path finding challenge datasets in Chapter 4. The real − world datasets were also used in Chapter 6 of this thesis.

SYNTHETIC DATASETS: The ESWC-2016 shortest path finding challenge provides both training and evaluation datasets:

- Training Dataset: The training data corresponds to the 10% transformation of the *DBpedia SPARQL Benchmark* [MLA+11] and comprises 9,996,907 triples in total. To evaluate our approach (i.e., in distributed settings), we divided this data into 4 equal parts, producing 4 different graphs, based on the order in which the triples were provided in the overall dataset. Based on our experience, the 4 datasets represent a reasonable trade-off between the size of the individual graphs and the overhead generated by communication. Table 7.6 presents the statistics about this data.
- Evaluation Dataset: This dataset corresponds to 100% of the DBpedia SPARQL Benchmark [3] and comprises 124,743,858 triples. We also divided this data into 4 different graphs. Table 7.7 presents statistics about the evaluation data.

Table 7.6: Training datasets statistics.

Dataset	Triples	Subject	Predicates	Objects
TDataset1	2925457	313036	7109	639743
TDataset2	2296074	92078	8536	1100275
TDataset3	2347536	95619	8135	1124389
TDataset4	2427840	97422	8267	1150104

Table 7.7: Evaluation datasets statistics.

Dataset	Triples	Subject	Predicates	Objects
EDataset1	30740699	680727	18541	10157624
EDataset2	31892427	723768	17001	9948302
EDataset3	30687403	686672	14119	9318528
EDataset4	31423329	674252	14175	9665835

REAL-WORLD DATASETS: As mentioned before, the real − world datasets are chosen from our previous evaluation and are provided by DisGeNET. DisGeNET is a platform of datasets containing one of the largest publicly available collections of genes and variants associated with human diseases. The datasets involved are: *Disease, hpoClass, doClass, phenotype, Protein, Variant, Gene, and pantherClass*. We chose these datasets because they are highly interlinked and contain shared resources. The data comprises of 7,265,423 triples. Table 7.8 shows the statistics for each of the individual datasets.

1 Datasets available from https://bitbucket.org/ipapadakis/eswc2016-challenge/downloads/
2 Datasets available from http://rdf.disgenet.org/download/v5.0.0/
3 http://benchmark.dbpedia.org/

Table 7.8: Real-world datasets statistics.

Dataset	Triples	Subject	Predicates	Objects
Disease	738626	60130	12	489756
doClass	101	21	11	63
Gene	1056346	119522	12	834502
hpoClass	253	36	11	151
pantherClass	272	40	9	123
Phenotype	83292	8441	8	66249
Protein	160537	14635	8	117034
Variant	5225996	708405	16	3628674

Queries

Along with datasets, the ESWC-2016 shortest path finding challenge also provides four different path queries with different `source` and `target` and K values. These queries are available from Task 1[4] of the challenge. In our evaluation, we used these four queries. For the *real-world* data, we used the 12 path queries used in the evaluation of Chapter 6. These queries were carefully chosen such that they show variation in terms of the number of possible paths between `source` and `target`, the number of distributed datasets contributing to the paths, and the length and complexity of the paths.

Evaluation Settings

In distributed computing, the network cost can be one of the key factors in the performance evaluation. For this reason we used two evaluation settings in our experiments:

- **Remote Setting With Network Cost:** To compare the distributed path finding systems, we loaded each RDF dataset (i.e., *synthetic, real-world*) into multiple Fuseki servers (version 1.3.0 2015-07-25T17) deployed on different physical machines with the following specifications: Ubuntu OS with 2.6GHz Intel Core i5 processors, 16GB 1600MHz DDR3 of RAM, and 500GB of storage capacity hard disks. It is important to note that each Fuseki server is integrated with our algorithm 7.2 and provides a public SPARQL endpoint to be queried remotely using SPARQL over HTTP requests.

- **Local Setting With Negligible Network Cost:** This represents the same settings as above, except that the four instances of the Fuseki server were started on the same machine with different ports for SPARQL endpoints. Since, all of the four instances of the Fuseki server were running on the same machine, the network cost is negligible.

Comparison

We compare our approach with state-of-the-art approaches that are able to find K paths in *RDF datasets*, loaded into triplestores with public SPARQL endpoints. Table 7.9 shows the existing approaches – including DpcLD – that we compared in this evaluation.

The performance metrics we used in our evaluation are: (1) the path computation time (in seconds), and (2) the memory consumption during the paths computation.

4 https://bitbucket.org/ipapadakis/eswc2016-challenge

Table 7.9: Systems that support RDF and path queries

System	Support		
	Single-graph	Distributed-graphs	K-Paths
DpcLD	✓	✓	✓
TPF [DVM16b]	✓	✓	✓
QPPDs[chapter 6]	✓	✓	✓
ESWC2016_winner [HSJ+16b]	✓	✗	✓

7.3.2 Performance Analysis

In this section, we compare the runtime performance of DpcLD, both with centralized and distributed path finding approaches for RDF datasets.

Comparison with Distributed Approach

As shown in Table 7.9, QPPDs and TPF are the two approaches that support finding the K paths in distributed *RDF datasets*. We compared DpcLD on exactly the same benchmarks used in the QPPDs and LDF evaluations.

DPCLD VS QPPDS In QPPDs's evaluation, the aforementioned 8 $real-world$ datasets were used. Each dataset was loaded into a dedicated Fuseki triplestore with a SPARQL endpoint. We used exactly the same settings in both approaches. Figure 7.3 shows the runtime comparison of our approach with QPPDs on 12 benchmark queries. As an overall performance evaluation, our approach is clearly faster than QPPDs on all of the 12 benchmark queries. The average (over all 12 queries) runtime of DpcLD is 4.0 seconds, while QPPDs took 17.2 seconds on average, leading to a performance improvement of greater than 400%. The main reason for this performance improvement is the lesser number of distributed path requests sent by our approach as compared to QPPDs. Furthermore, DpcLD makes use of the cache to avoid sending duplicate requests while that is not controlled by QPPDs.

DPCLD VS TPF We compared our approach to the results presented in the paper about TPF [DVM16b]. To have a fair comparison, we used machines for DpcLD with the same specifications as was used for the evaluation of TPF. Figure 7.4 depicts the comparison between DpcLD and the TPF approach in terms of response time. We outperformed TPF by several orders of magnitude. In general, the TPF servers only perform executing single triple pattern SPARQL queries. The load of the query execution is distributed among TPF client and server, thus ensuring high availability with a slight performance loss.

Comparison with Centralised approaches

In this section, we compare our distributed DpcLD approach with a centralized path finding solution over RDF data. The goal of the comparison is to show how DpcLD scales as compared to centralized solutions. To this end, we compare DpcLD with the winner of the ESWC2016 challenge based on the datasets and queries used in the ESWC2016 challenge, i.e, the synthetic datasets (both training and evaluation) and the four queries Q1-Q4. We used both of the aforementioned evaluation settings (i.e., remote and local settings).

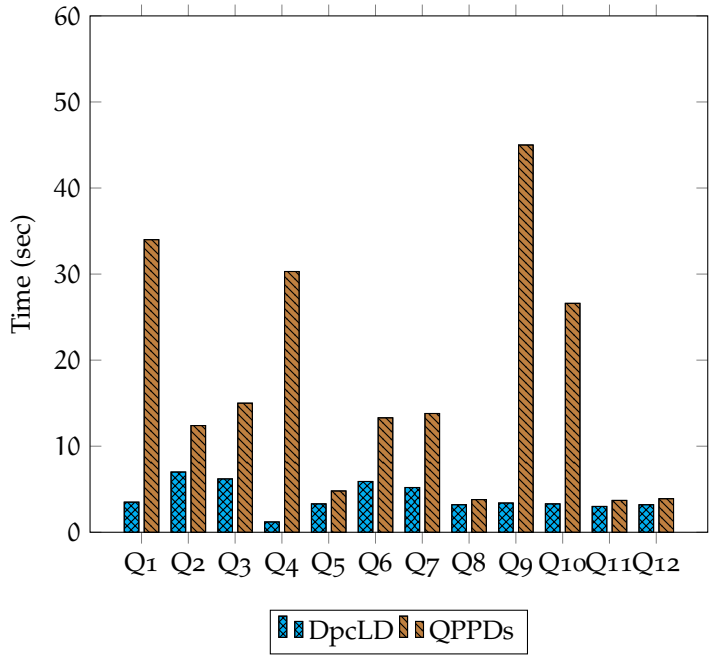


Figure 7.3: Response time per query with DpcLD vs. QPPDs.

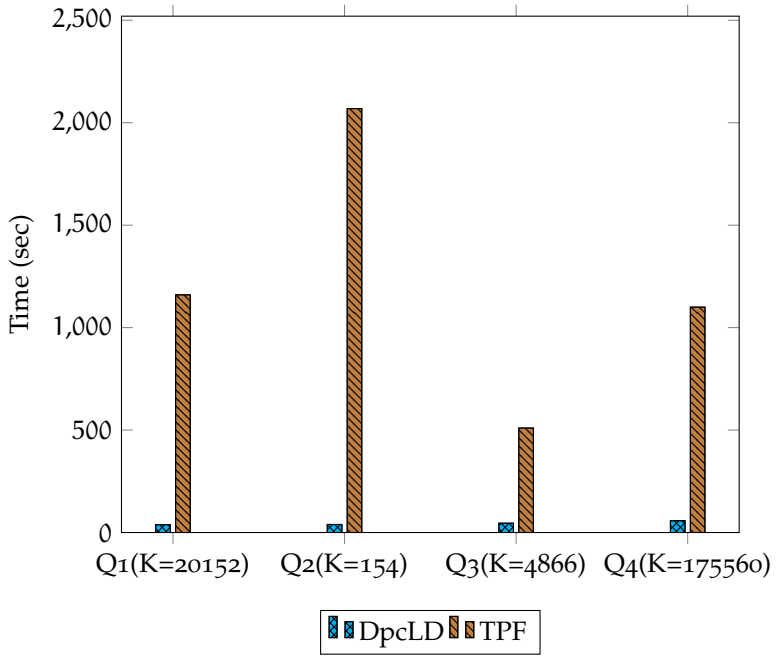


Figure 7.4: Response time per query with DpcLD vs. TPF.

TRAINING DATA RESULTS Figure 7.5 shows the comparison of the ESWC2016_winner with DpcLD when deployed both in local and remote settings. The results clearly suggest that the centralized solution is much faster for smaller K values. However, its performance significantly drops when we increase the required K number of paths. On the other hand, the performance of DpcLD is rather slow for smaller K. However, it scales better as compared to ESWC2016_winner as we increase K. The reason DpcLD performs slower with smaller K values is that it first collects the partial paths and then performs the processing. The results also suggest that DpcLD

is approximately 4 times slower in remote settings as compared to the local settings. This means that network costs play an important role in the performance of DpcLD.

The runtime performance results shown in Figure 7.5 are highly correlated to the memory consumption by each of the systems shown in Figure 7.6. In general, DpcLD consumes less memory resources when deployed in remote settings, followed by DpcLD in local settings, and then ESWC2016_winner. The results suggest that the significant performance drop in ESWC2016_winner is due to the large amount of memory consumed, as we increase the value of K. On the other hand, memory consumption in DpcLD is more controlled as compared to ESWC2016_winner for large K values.

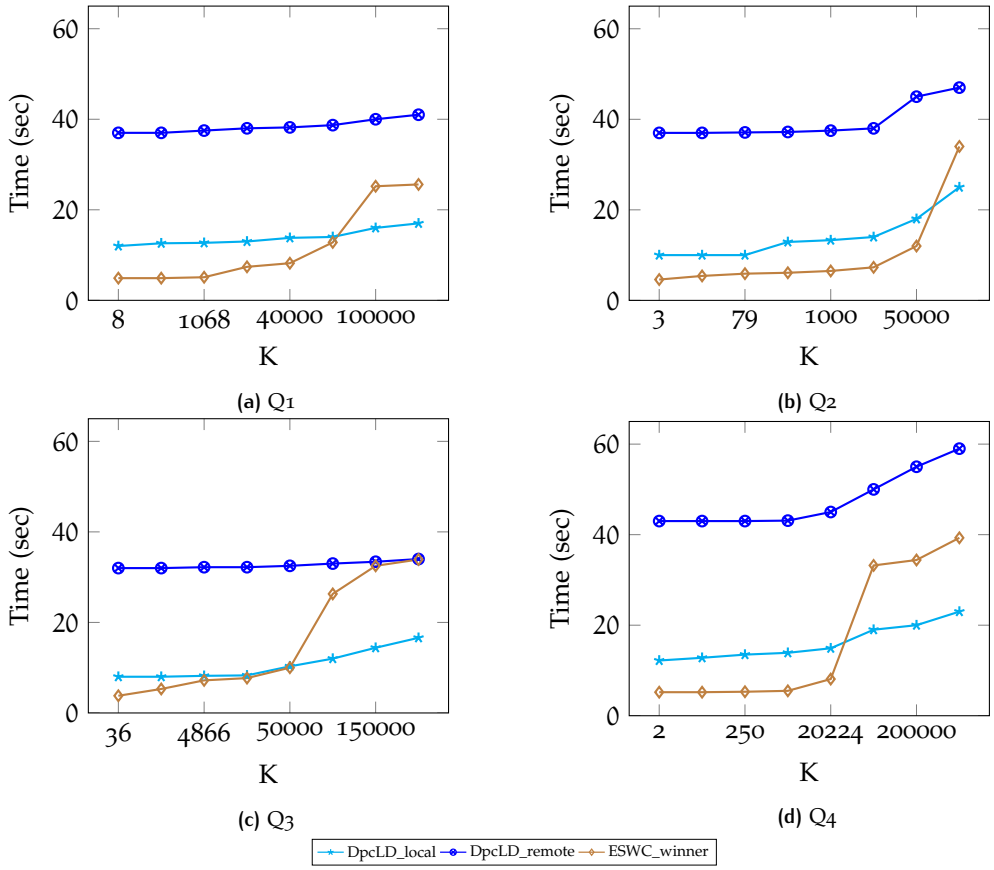


Figure 7.5: Response time per query for DpcLD vs. ESWC2016_winner.

Figure 7.6 shows the memory consumed by each system. We noticed that DpcLD outperformed ESWC2016_winner in all queries except for one case (i.e., K =250) in query Q4. This shows on both the four remote instances and the DpcLD engine deployed locally.

EVALUATION DATA RESULTS Figure 7.7 shows the runtime performance for the bigger dataset (i.e, 100% DBpedia) as compared to the training dataset (i.e, 10% DBpedia). In general, it can be seen that the performance of ESWC2016_winner degrades much faster with values of K as compared to the training data. On the other hand, the performance graph of DpcLD is almost linear with the increase of K values, confirming that DpcLD scales better as compared to ESWC2016_winner. For Q1 we set the maximum $K = 400,000$, where ESWC2016_winner was only able to retrieve paths up to $K = 250,000$. After that limit, it started sending *out of memory* exceptions. DpcLD, on the other hand, was able to retrieve paths to the maximum K limit. For

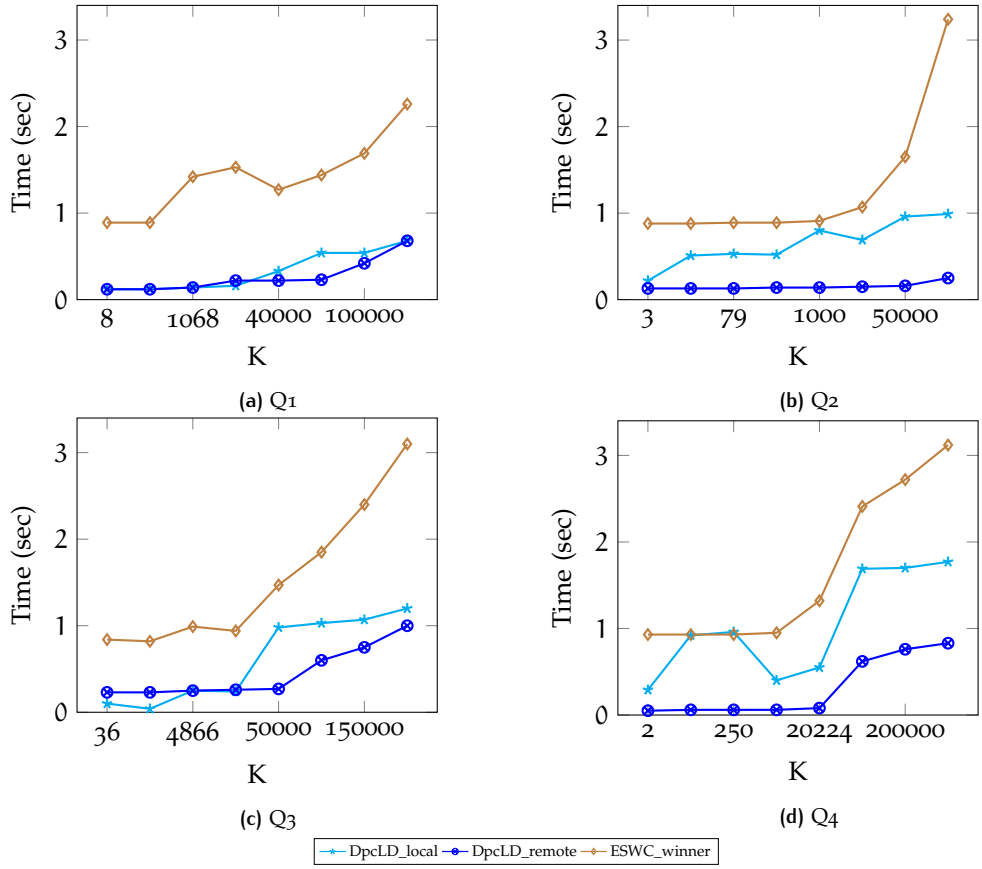


Figure 7.6: Memory consumption per query for FedS vs. ESWC2016_winner.

query Q2, it can be seen that DpcLD performed better than ESWC2016_winner throughout from the minimum to the maximum values of K. For queries Q3 and Q4, ESWC2016_winner outperformed DpcLD up to a certain path limits (i.e, around $K = 100,000$). However, it started sending *out of memory* exceptions after that limit. DpcLD was able to retrieve paths up to the highest tested values of K for both queries Q3 and Q4. The results suggest that with huge volumes of data and higher K values, the algorithms that work on a single graph may face performance issues or *out of memory* exceptions. This comparison shows that DpcLD could be applicable as an alternative choice to local traversal approaches when it comes to dealing with large amount of data. Figure 7.8 shows the memory consumed by each system over the evaluation data. In general, we noticed that DpcLD required about 8 times less memory than ESWC2016_winner. As already mentioned, ESWC2016_winner throws *out-of-memory* exceptions for Q1, Q3 and Q4 over large values of K. This shows that DpcLD can be a good candidate when finding paths over large distributed datasets using machines with low memory resources.

Randomized data and DpcLD performance

In this section, we show the result of testing DpcLD when the dataset is randomly distributed, rather than being divided in the order provided. In the original dataset, the triples are provided ordered by subject, meaning that triples with the same subject are more likely to end up in the same partition. The objective here is to show the extent of the impact of such a distribution on the performance of DpcLD.

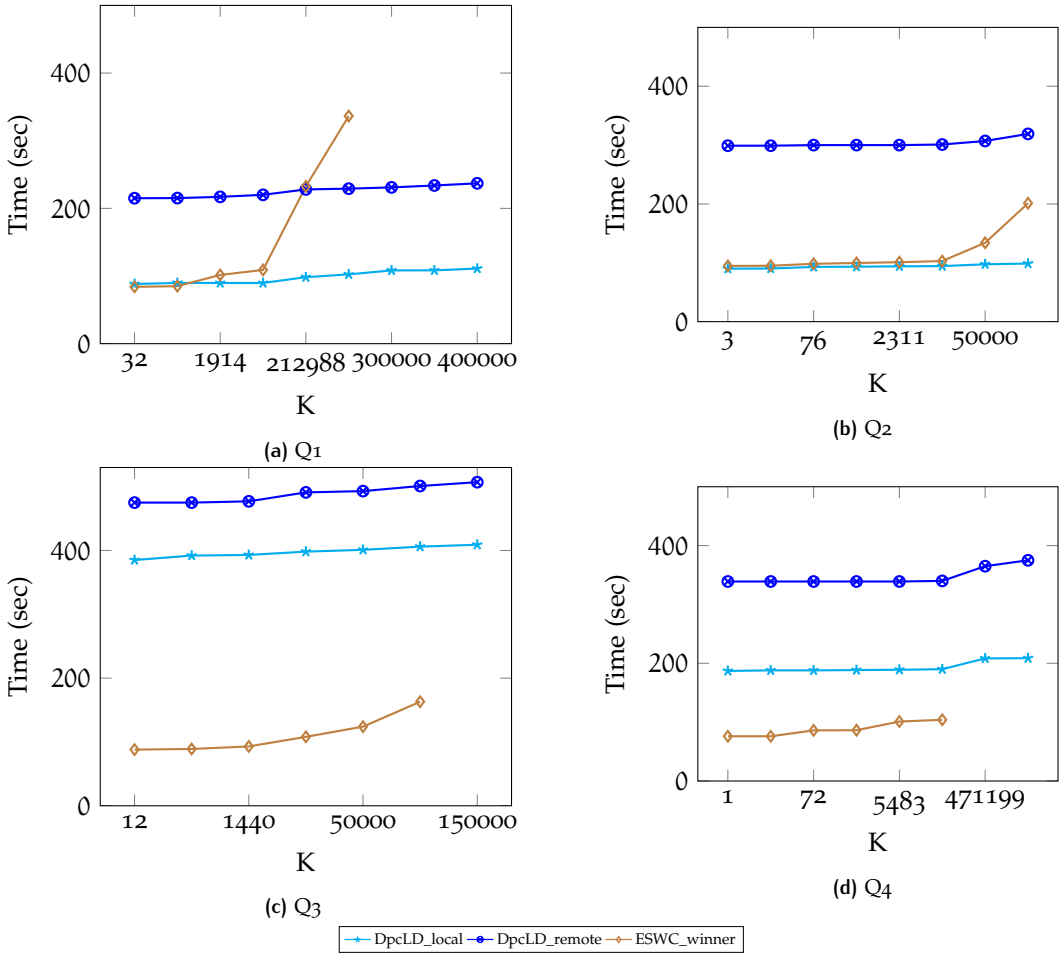


Figure 7.7: Response time per query over evaluation dataset for DpcLD vs. ESWC2016_winner.

We randomized the training data using the *shuf* command[5], partitioned it as before, and loaded it into 4 remotely running Fuseki servers. This represents an extreme case, unlikely to be encountered in practice, where triples are distributed in a way that does not follow any specific pattern. It therefore represents a "worst case scenario" for DpcLD, as a way of *torture testing* the system.

Figure 7.9 shows the response time of DpcLD to compute answers for each of the queries Q1-Q4 with the indicated K values on the training dataset (10% DBpedia). Compared to the results obtained with the same K values as depicted in Figure 7.4, the effect of random data distribution becomes obvious. This emphasises how distributed approaches like DpcLD do rely on data being distributed in a way that is meaningful and that supports balancing local and remote computations. It is worth mentioning here that QPPDs failed to respond when tested on the same setting, and that results are not available with randomised datasets for TPF.

5 https://shapeshed.com/unix-shuf/

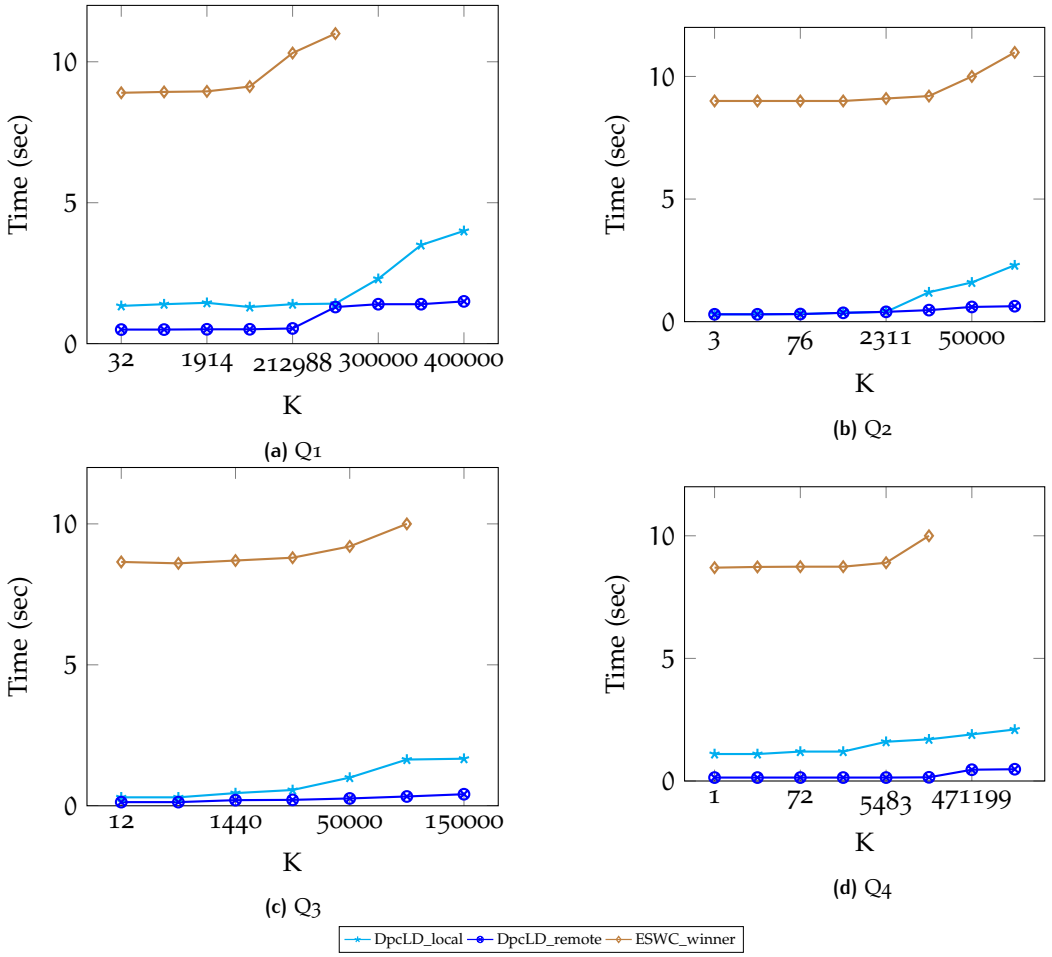


Figure 7.8: Memory consumption per query over evaluation dataset for DpcLD vs. ESWC2016_winner.

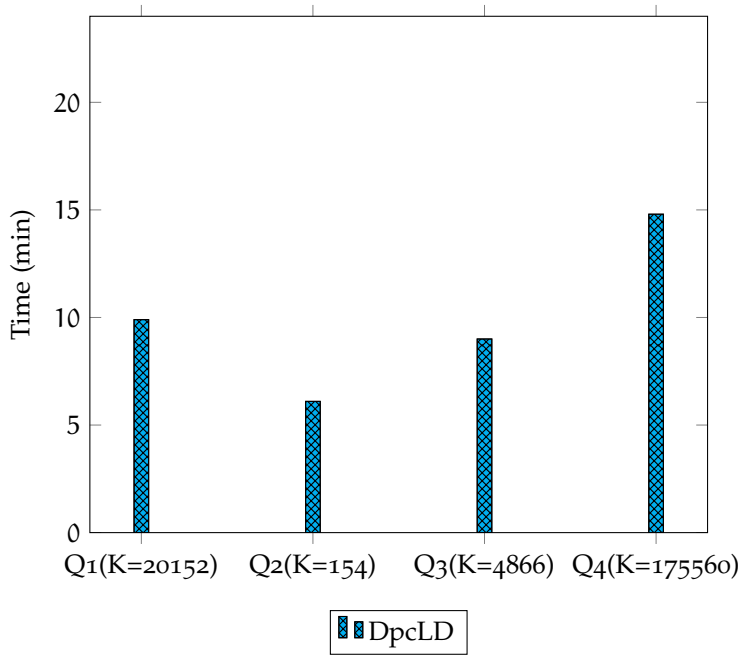


Figure 7.9: Response time per query from DpcLD with randomized dataset.

7.4 CONCLUSION

In this chapter, we propose DpcLD, an engine for finding paths in distributed RDF datasets exposed as SPARQL endpoints. DpcLD is an index-free approach and hence does not require any pre-computations like QPPDs (presented in Chapter 6) for generating the index. Since DpcLD does not require an index, it has the capability of querying up-to-date data. DpcLD exploits and aggregates partial paths within a distributed environment to retrieve the required K paths. We used both synthetic and real-world datasets and compared the performance of DpcLD with distributed and centralized path finding approaches over RDF datasets. Furthermore, we used two different evaluation settings to measure the effect of the network cost on the runtime performance. The results suggest that DpcLD outperforms other distributed methods and scales better as compared to the best centralized approach for path retrieval over RDF datasets. The results suggest that DpcLD, when tested for local computation, could be applicable as an alternate choice to local traversal approaches when it comes to deal with large amount of data with less memory.

8 CONCLUSIONS

As we discussed previously, the large collection of interlinked data has created new challenges in the context of data management, knowledge discovery, interoperability, and integration. These challenges have led academia and industry to pay more attention towards efficiently storing, traversing and querying such enormous, heterogeneous and information-rich knowledge graphs. We have tried to address some of these challenges and answer the research questions mentioned in Chapter 1. We wrap up our work with a discussion on the approaches proposed in this dissertation to address those challenges. The work presented in this thesis was divided into two parts: The first part discussed centralised approaches and the second highlighted and provided knowledge about the different approaches we proposed for distributed systems. In the following, we present our conclusions on each approach. Finally, we present an overall contribution and highlight the future direction for research in this area.

8.1 CONTRIBUTIONS

This section provides a summary of the contributions discussed in chapters 4, 5, 6 and 7 of this thesis.

8.1.1 Centralized approach

CHAPTER 4 TOP-K-PATH: The work presented in chapter 4 is the first part of this thesis which presents work related to querying paths in a centralized data management scenario. Chapter 4 reported on an extension of SPARQL1.1 property paths that allows to compute and return the K shortest paths matching a property path expression between source and target nodes.

We highlighted that the current SPARQL1.1 Property path features are not sufficient for graph analytics. We also discussed the state-of-the-art works and their limitations in the context of path queries (see Section 3.2 of Chapter 3).

To fulfill the research gap and to answer our first research question, we proposed an efficient solution for the k shortest path problem in the context of SPARQL. Our solution is based on the adaptation of bidirectional breadth first search to top k path computation, extending it with path expressions and embedding it into Jena ARQ via the extension mechanism of property functions. We used indexed HDT backend data, and our implementation demonstrates very promising performance over large-scale graphs, even without complex optimizations. Our proposed solution outperformed the state-of-the-art systems.

8.1.2 Distributed approaches

Although we demonstrated promising results for *centralized* RDF graphs, Linked Data by nature is distributed and most of the approaches (including ours summarised above) working in a centralized way (i.e., only on single graphs) would require, to be applicable in this context, that all sources are first merged together, which can create issues of scale and in keeping the data up-to-date. One of the other known issues that make centralized approaches inadequate in this context is the massive amount of data generated by different organisations or vendors accessible only through their respective endpoints. For instance, in the biomedical domain, it is often the case that the data representations of two biological entities are scattered across different datasets distributed over some network. To overcome the limitations existing in centralized approaches, we also proposed and extended our work for distributed paths. In the following, we discuss the contributions from each of our distributed approaches.

CHAPTER 5 FEDS: In Chapter 5 we proposed FedS, a path traversal approach that works in a P2P network and that not only tackles the limitations mentioned above, but also the existing drawbacks (highlighted in the motivation, Section 5.1) in P2P systems. Our initial evaluation results are encouraging, with FedS retrieving all paths in comparable query processing times when compared to state-of-the-art triplestores. However, in our current implementation, FedS also has some limitations which are discussed in Section 5.6 of Chapter 5.

The approach we presented in Chapter 5 does not follow the core P2P overlay network architecture or DHT algorithms (e.g., chord or pastry), and is also not bounded by restricted rules or sophisticated data distribution mechanisms as is required in standard P2P architectures. Our proposed solution, even without optimisation, was comparable with the Virtuoso and Blazegraph triplestores.

CHAPTER 6 QPPDS: The FedS approach we presented in Chapter 5 belongs to the homogeneous networks paradigm (as discussed in Section 2.4.1 of Chapter 2) and works similarly to a distributed system in a controlled environment. As Linked Data by nature is distributed, in a fully controlled architecture, we may not be able to harness the full benefit of distribution. Consequently, in Chapter 6 we propose a federated path query engine called QPPDS that can federate path queries to triplestores (e.g., Stardog [Sir]) that support path queries and are distributed over the Linked Data cloud. The concept behind the QPPDs approach is tackling those challenges that were described for centralised path querying in Chapter 4 and for homogeneous approaches that work in controlled environments in Chapter 5.

We have noticed that different database systems, not necessarily focusing on RDF, have already started supporting the enumeration of paths beyond the current SPARQL 1.1 Property Path specification. These include the Stardog [Sir] system and the systems mentioned in Table 3.1 (see Column "Output Paths") of Chapter 3. The current implementation we provided here is tested on the Fuseki server [Jena] running with a common Breadth First Search (BFS) algorithm. QPPDs could however work with such systems that provide path enumeration "off-the-shelf" and could be easily adapted for systems relying on different graph models (e.g., weighted, unweighted, property, etc.)

Another salient feature, which is not available in other state-of-the-art approaches, is that QPPDs provides a dataset, exposed through a SPARQL endpoint, of paths that contain statistical and provenance information for each path.

CHAPTER 7 DPCLD: As described above, QPPDs has a number of advantages over FedS. This is due to QPPDs being designed to work in a heterogeneous environment. On the other hand, FedS was implemented as a P2P approach that was able to work only in a controlled environment. With all its advantages over FedS, the QPPDs approach however depends on a precomputed index, and due to this dependency, result completeness is not assured if the data is updated after the index is computed. Consequently, in Chapter 7, we propose an index-free distributed path query engine called DpcLD that has the capability of querying paths even if the data is updated in remote endpoints. DpcLD exploits and aggregates partial paths within a distributed environment to retrieve the required K paths. We used both synthetic and real-world datasets and compared the performance of DpcLD with distributed and centralized path finding approaches over RDF datasets. Furthermore, we used two different evaluation settings to measure the effect of the network cost on the runtime performance. The results suggest that DpcLD outperforms other distributed methods and scales better than to the best centralized approach for path retrieval over RDF datasets.

The results suggest that DpcLD, when tested for local computation, could be applicable as an alternative to local traversal approaches when it comes to dealing with a large amount of data within an environment with limited memory capacity.

8.1.3 Overall Contribution

Graph analysis is one of the core research topics in relation to graph databases. Many applications in different domains, including social networks or biological networks, fundamentally rely on graph data and on their ability to extract meaningful knowledge from those graphs.

One of the important features in graph analytics is pathfinding. In this thesis, we proposed different approaches to address the pathfinding task especially in a distributed environment. From the individual contributions described above, the overall conclusion of this work is that graph management and distribution have a strong impact on the efficiency and accuracy of pathfinding, and therefore that different solutions are required depending on their characteristics.

The first important problem (see Research Question Q1 in 1.3) we addressed by reviewing the stat-of-the-art was the limitations in the context of expressiveness of the navigational queries as well as the query performance. We proposed that large-scale graphs can be queried efficiently, and we concluded that compressed data techniques are more cost effective in terms of query performance and take less disk space compared with approaches that use raw data. Moreover, we found that there exist important properties in RDF data that can not be captured with the current SPARQL query language. Consequently, we proposed that the current SPARQL query language can be equipped with more interesting features, particularly for graph analytical purposes. We extended the SPARQL language for property path queries in a more expressive way, while keeping consistent with the syntax of SPARQL, so the results obtained can be more human-understandable. We argue that our proposed extension can enhance the graph analysis feature in the current specification of SPARQL1.1 property paths.

In the second part of this thesis, we addressed the problem related to the distributed processing (see Research Question Q2 in 1.3) of the navigational queries. We demonstrated how efficiently data can be queried when the distribution of data is either customized in a controlled environment or even if the data is accessible publicly. By reviewing the state-of-the-art work, we found that there is a research gap for the distributed navigational query processing. For instance, currently most of the existing approaches can only be applicable in homogeneous systems and are purely developed for customised data distribution or for controlled environments. For example, P2P-based and Hadoop-based approaches are considered to be homogeneous. In contrast to these approaches, we proposed hybrid approaches in chapters 5 and 7. In our approaches we do not need to follow the standard and complicated architecture of the network (i.e., P2P) or the management of the ecosystems (i.e., Hadoop, etc). Our proposed solutions are "plugin based" and easy to adopt. Furthermore, we introduced the concept of source selection which improved path query performance by selecting only relevant datasets. We also tested one of our approaches called DpcLD 7 after randomising data distribution. In the original data, triples were provided in such a way that triples with the same subject were more likely to end up in the same partitioned dataset. The objective of randomising distribution was to test the impact on query performance of random data distribution. We argue that such distributions can be used as a "worst case" benchmark for different federated query systems. This aspect is especially important in heterogeneous systems where there is no centralised control over the distribution of the data, as in the Linked Data Cloud for example. This motivated the approach presented in Chapter 6 working over existing, distributed endpoints that support paths and enumeration "off-the-shelf". Because the data comes from heterogeneous sources and their distribution has been shown to affect path querying, providing context for the result of path queries becomes critical. QPPDS therefore provides provenance information about paths and their relevant datasets which we argue is a salient feature that can be useful to create a dataset of path-related information.

8.2 FUTURE WORK

In this thesis, we studied different aspects of path querying over large-scale as well as distributed graphs. While we tried to address the challenges in graph analytics both in central and distributed paradigms, there are still further questions which remain open and can be explored.

The work we presented in Chapter 4 proposes an important feature called $\mathrm{Top}-\mathrm{K}$ path. We proposed this work as an extension to the current SPARQL1.1 property paths. Here we meant for $\mathrm{Top}-\mathrm{K}$ to be based on sorting paths on their length. However, there are other features that are relevant to the analysis of the graph which can be added to our current implementation. Those features can look at $\mathrm{Top}-\mathrm{K}$ paths based on ranking, for instance, using the "importance", "relevance", or other characteristics of the paths. Also, the current implementation was tested only on Fuseki, and does not provide a visual interface. Implementations that rely on other triplestores and through which path exploration can be visualized could also be considered.

In addition, we already have introduced different source selection mechanisms for distributed path queries in our approaches. Further research can focus on optimised source selection (e.g., based on ranking candidate sources) to choose the relevant data sources. For example, our source selection mechanism in Chapter 6 is based on indexing, where we index two datasets based on their "common" nodes. However, we could also consider a heuristic indexing approach

to reduce the number of remote requests sent. The current implementation does not employ any cache mechanism, hence we believe that a disk-based cache can improve the query performance.

As shown in Chapter 7, currently our DpcLD engine communicates only with a "shared" algorithm running on remote endpoints and answering local path queries. However, with some modifications, databases that support SPARQL1.1 property paths (e.g., Virtuoso, Blazegraph, etc) could be adapted to achieve the same results, reducing the need for a specific shared algorithm. The extent of those adaptations and whether reference implementations could be provided for them is an interesting area to further explore.

Finally, there are a number of aspects of our implementations that could be improved, including:

- Our current implementations for distributed approaches work only with queries considering arbitrary paths between source and target nodes, and could be extended to also include regular expression-based filtering.
- A shared, established benchmark for testing distributed SPARQL property paths would help in making results more comparable and reproducible.
- In many domain, resources can be private and access to those resources can be tightly controlled by the resource providers. Consequently, privacy can be a major concern when accessing data from those resources. Currently none of the existing approaches to distributed path querying (including ours) considered this important aspect. Thus, a secure and access-controlled based path computation over distributed datasets can be explored.
- All implementations have currently been tested on RDF graphs. However, all algorithms presented in this thesis can easily be adapted for other graph models (e.g., weighted, unlabelled, etc).

In this thesis, we focused only on Linked Data and provided different solutions for directed, labeled graph data models. However, data retrieval systems are facing a paradigm shift due to the proliferation of specialized data storage engines (SQL, NoSQL, Column Stores, MapReduce, and different Graph models) supported by varied data models (CSV, JSON, RDB, RDF and Property graphs, etc). Although, steps have been taken towards query answering over multiple data models [KZJ+19], at the time of writing this thesis, we could not find any work addressing distributed *navigational* queries in the context of multiple data models. We believe that addressing this gap is an interesting research question, expending the practical applicability of path queries to polystores.

Appendices

A DEFINITIONS FOR LANGUAGE FEATURES

This appendix includes the dictionary of features used to compare RDF query languages in Section 3.1 of Chapter 3. The features presented in Table 3.1 are inspired by numerous previous studies such as [Prz17; Abi97; HBE+04; AG05; AG08; VCM08].

NAMESPACES: distinguish between multiple RDF datasources, hence, are an essential part of pattern-based query languages to query the corresponding dataset. This way, instead of writing a complete URI in a query pattern, we can shorten the entity URIs. For instance, SPARQL queries presented in this thesis use namespaces (see for instance the query 11).

CONSTRAINTS: reflects the ability to express restrictions on the RDF data while constructing a query. For instance, SPARQL uses the FILTER clause to apply constraints or restrictions on any resource.

LANGUAGE: is the syntax used to construct queries that communicate with the corresponding database . For example, XPath, SPARQL, etc. are the query languages understandable by their respective databases to fetch or modify data.

LEXICAL & VALUE SPACES: are the types applied on different values to retrieve. For example, in SPARQL, XML Schema provides different typed literals, such as "string", "integer", etc. These are *the lexical spaces* that tell and distinguishe different types. For example, "5" and 5 are considered not equal if defined as belonging to different types.

GRAPH PATTERNS: were first introduced in 1997 as *path expressions in patterns* and were considered crucial for querying semistructured data in early works [Abi97]. By concatenating different entities or attributes they typically correspond to the definition of a graph traversal process.

ADJACENT RESOURCES: basically represent "neighbours" and are fundamental in navigational query languages. In graph data, two resources can be adjacent either by *incoming* or *outgoing* edges.

ADJACENT PREDICATES: are considered analogous to the previous definition. In a path query context, one can ask for all edges related to a given resource. For instance, a user can be interested in calculating the outgoing or incoming edges for a certain resource.

FIXED-LENGTH PATHS: are advanced features in different queries and approaches in the context of graph analysis. A user can ask to find the path between two resources as connected by a path of a given length (a.k.a fixed hop path).

PATH VARIABLES: have been introduced in many systems proposed recently (e.g., Stargdog [Sir], Top-k-Path [SMU+17], RDF3X_paths [GN11], etc). The idea is that, instead of the atomic representation of each resource, a path variable holds a complete chain of resources contributing to a certain path.

INVERSE PATH: in a traversal mechanism where a path is not only checked forward (subject→object) but also backward (subject←object).

NON-SIMPLE PATHS: are the paths where multiple occurrences of a same edge and node can happen. In RDF terminology, arbitrary resources are allowed to be traversed multiple times. However, such paths also come with cycles which may create a problem while traversing if not handled.

RECURSIONS (REGULAR EXPRESSIONS): for capturing paths of arbitrary length in a transitive way, recursions are the building blocks. For instance, *friends of a friend*(FOAF) relations are treated as recursive connections and are transitively computed for a given query.

CONSTRAINED REGULAR EXPRESSIONS: are used to apply filters on regular expressions such that the paths is required to satisfy the applied constraints.

OPTIONAL PATTERNS: are applied on highly diversified semi-structured data which might be incomplete or irregularly retrieved for a given query. For that, optional patterns enable queries to add results only if present.

ENTAILMENT (REASONING): With this feature, information can be inferred and can be added to the data if not explicitly present. In the context of RDF, OWL is a language used to infer relations using entailment rules. Different *reasoners* are used to transitively compute and infer such relations. However, in the case where there is no need to use all resources, for instance, in large knowledge graphs, it is good to equip the query language with reasoning capabilities.

QUERYING TOPOLOGY: In the context of *entailment* (Reasoning), multiple approaches are used to handle data. The approaches , in particular the ones based on traditional graph data, have different model representations as compared to the RDF graph model. For example, in RDF, the same resources can be modeled in predicate and subject/object positions, which in general is not possible for other graph models. Therefore, the term querying topology was used in [Prz17] for the query languages on the basis that they "(1) capture *all* querying facilities inherent to RDF triple model, and (2) facilitate smooth integration of schema and data in a combined fashion".

OUTPUT PATHS: the interpretation of traversed paths according to [Abi97] can be divided into two notions: (1) the set of objects that also includes the end and start nodes of a path and (2) the paths themselves. A significant impact on path complexity evaluation depends on which one of the two is chosen, since computing and enumeration of complete paths is a costly task. This is why most of the existing query languages, for instance, SPARQL, only produce the nodes involved in paths but do not enumerate the paths themselves. In contrast to those approaches

and systems, the approaches that produce complete paths are considered to support this feature. One of such approaches is presented in chapter 4.

DEGREE OF RESOURCE: is a scalar value. The number of adjacent edges (introduced earlier) determines this value [HBE+04; AG05].

DISTANCE BETWEEN RESOURCES: is the total number of resources involved to find the path between two given resources.

SHORTEST PATH: among all traversed paths is the path that has the least number of resources included between the source and target nodes. There are different kinds of shortest path problems, for instance *single-source-single-destination shortest path (SSSP)*, *all-pair shortest path (APSP)*, etc.

UNION: is a basic algebraic operation where two sets are merged into one.

DIFFERENCE: is a basic algebraic operation where one set is subtracted from the other one.

AGGREGATION: an aggregated value of multiple resources or literals.

SORTING: sorting results in accordance with lexical or value-space order.

CLOSURE: is a property that represents the results of an operation produced again in the same model. For instance, if the input data for a query is RDF, the results should also be RDF data [Prz17].

COLLECTIONS AND CONTAINERS: in RDF terms, a group of resources can be represented as a container or collection. For instance, rdf:Seq is a collection representation of resources in human-readable numerical order.

B EXPERIMENTAL QUERIES

Following are the prefixes used throughout in different queries:

```
PREFIX : <http://insight-centre.org/sample/>
PREFIX ppfj: <java:org.centre.insight.property.path.>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>
```

The following queries are some of the experimental queries used in the first part of this thesis in Chapter 4.

```
SELECT *
WHERE { VALUES(?s ?o){(dbr:1952_Winter_Olympics dbr:Elliot_Richardson)}.
        ?path ppf:topk (?s ?o 2) }
```

```
SELECT *
WHERE { VALUES(?s ?o){(dbr:1952_Winter_Olympics dbr:Elliot_Richardson)}.
?path ppf:topk (?s ?o 2 (:after/:after/:officiallyOpenedBy/:defense)*) }
```

```
SELECT *
WHERE { VALUES(?s ?o){(dbr:1952_Winter_Olympics dbr:Elliot_Richardson)}.
?path ppf:topk (?s ?o 2 (:after/:after/:officiallyOpenedBy/!(:defense))) }
```

```
SELECT *
WHERE { VALUES(?s ?o){(dbr:1952_Winter_Olympics dbr:Elliot_Richardson)}.
?path ppf:topk (?s ?o 2 (:after/:after*/:officiallyOpenedBy/!(:defense))) }
```

```
SELECT *
WHERE { VALUES(?s ?o){(dbr:1992_Spanish_Grand_Prix dbr:Michael_Schumacher)}.
?path ppf:topk (?s ?o 1) }
```

```
SELECT *
WHERE { VALUES(?s ?o){(dbr:1952_Winter_Olympics dbr:Elliot_Richardson)}.
```

```
?path ppf:topk (?s ?o 2 ((:after/:after/:officiallyOpenedBy/:defense )
| (:after/:after/:officiallyOpenedBy/!:justice))) }
```

In the rest of this thesis we used a query template proposed in chapter 4. In contrast to the above queries where queries were constructed based on the path filters, here in the second part of this thesis we used more complex queries, using a query template without any path filter which finds arbitrary paths consisting of any predicate (i.e., (: |! :)* or "a wildcard"). The following query template in Listing 17 is used with different input parameters shown in different tables corresponding to each chapter.

```
SELECT * WHERE { ?path ppfj:topk (<:source>  <:target>  K) . }
```

Listing 17: Query template used throughout the second part of the thesis.

The following table (B.1) shows the input parameters for each query used in chapters 6 & 7. Datasets used for these queries contain real-world data and are downloaded from RDF-DisGeNET[1].

Table B.1: Input parameters as source and target for query template shown in Listing 17 .

Source node	Target node
http://purl.obolibrary.org/obo/HP_0000818	http://semanticscience.org/resource/SIO_000275
http://purl.obolibrary.org/obo/HP_0004942	http://www.human-phenotype-ontology.org/hpoweb/showterm?id=HP:0001626
http://purl.obolibrary.org/obo/HP_0004942	http://bioportal.bioontology.org/ontologies/DOID/DOID:7
http://identifiers.org/doid/DOID:0014667	http://semanticscience.org/resource/SIO_000275
http://identifiers.org/dbsnp/rs769022521	http://identifiers.org/ncbigene/10128
http://purl.obolibrary.org/obo/HP_0004942	http://rdf.disgenet.org/v5.0.0/void/doClass
http://linkedlifedata.com/resource/umls/id/C0033581	http://purl.obolibrary.org/obo/HP_0000024
http://identifiers.org/dbsnp/rs769022521	http://monarchinitiative.org/gene/NCBIGene:10128
http://purl.obolibrary.org/obo/HP_0000818	http://linkedlifedata.com/resource/phenotype/id/HP:0000818
http://purl.obolibrary.org/obo/HP_0000818	http://bio2rdf.org/umls:C4025823
http://identifiers.org/dbsnp/rs769022521	http://semanticscience.org/resource/SIO_000275
http://identifiers.org/dbsnp/rs769022521	http://rdf.disgenet.org/v5.0.0/void/pantherClass

The following table (B.2) shows the input parameters for the training and evaluation datasets used in Chapter 7.

1 http://rdf.disgenet.org/download/v5.0.0/

Table B.2: Input parameters as source and target for query template shown in Listing 17 .

Source node	Target node
http://dbpedia.org/resource/Felipe_Massa	http://dbpedia.org/resource/Red_Bull
http://dbpedia.org/resource/1952_Winter_Olympics	http://dbpedia.org/resource/Elliot_Richardson
http://dbpedia.org/resource/Karl_W._Hofmann	http://dbpedia.org/resource/Elliot_Richardson
http://dbpedia.org/resource/James_K._Polk	http://dbpedia.org/resource/Felix_Grundy

C PATH DATA MODEL

The following figure shows the schema model used for generating the provenance based information about the paths retrieved and stored in a SPARQL endpoint as discussed in Chapter 6.

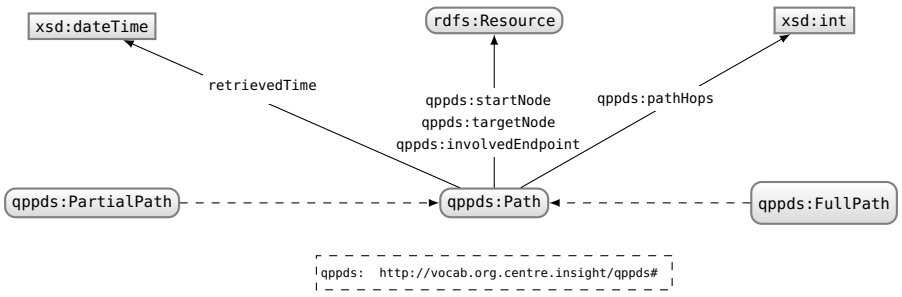


Figure C.1: QPPDS data model (dashed lines indicate sub-classes).

BIBLIOGRAPHY

[ABE08] F. Alkhateeb, J.-F. Baget, and J. Euzenat. "Constrained regular expressions in SPARQL". In: *International conference on semantic web and web services - SWWS*. 2008, pp. 91–99.

[Abi97] S. Abiteboul. "Querying semi-structured data". In: *International Conference on Database Theory*. Springer. 1997, pp. 1–18.

[ABK+07] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. "Dbpedia: A nucleus for a web of open data". In: *The semantic web*. Springer, 2007, pp. 722–735.

[ACH+04] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, and T. Van Pelt. "Gridvine: Building internet-scale semantic overlay networks". In: *International semantic web conference*. Springer. 2004, pp. 107–121.

[ACP12a] M. Arenas, S. Conca, and J. Pérez. "Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard". In: *Proceedings of the 21st international conference on World Wide Web*. ACM. 2012, pp. 629–638.

[ACP12b] M. Arenas, S. Conca, and J. Pérez. "Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard". In: *Proceedings of the 21st international conference on World Wide Web*. 2012, pp. 629–638.

[AG05] R. Angles and C. Gutierrez. "Querying RDF data from a graph database perspective". In: *European semantic web conference*. Springer. 2005, pp. 346–360.

[AG08] R. Angles and C. Gutierrez. "Survey of graph database models". In: *ACM Computing Surveys (CSUR)* 40.1 (2008), pp. 1–39.

[AGH04] R. Angles, C. Gutierrez, and J. Hayes. "RDF query languages need support for graph properties". In: *Technical Report tr/dcc-2004–3* (2004).

[AKC+19] K. S. Aggour, V. S. Kumar, P. Cuddihy, J. W. Williams, V. Gupta, L. Dial, T. Hanlon, J. Gambone, and J. Vinciquerra. "Federated Multimodal Big Data Storage & Analytics Platform for Additive Manufacturing". In: *2019 IEEE International Conference on Big Data (Big Data)*. IEEE. 2019, pp. 1729–1738.

[AMS07] K. Anyanwu, A. Maduko, and A. Sheth. "Sparq2l: towards support for subgraph extraction queries in rdf databases". In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 797–806.

[ASM20] J. Aimonier-Davat, H. Skaf-Molli, and P. Molli. "How to execute SPARQL property path queries online and get complete results?" In: *4rth Workshop on Storing, Querying and Benchmarking the Web of Data (QuWeDa 2020) Workshop at ISWC2020*. 2020.

[AVL+11] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. "ANAPSID: an adaptive query processing engine for SPARQL endpoints". In: *International Semantic Web Conference*. Springer. 2011, pp. 18–34.

[BCF+06a] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. "Using partial evaluation in distributed query evaluation". In: *Proceedings of the 32nd international conference on Very large data bases*. 2006, pp. 211–222.

[BCF+06b] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. "Using partial evaluation in distributed query evaluation". In: *Proceedings of the 32nd international conference on Very large data bases*. 2006, pp. 211–222.

[BDM+94] T. Berners-Lee, D. Dimitroyannis, A. J. Mallinckrodt, and S. McKay. "World Wide Web". In: *Computers in Physics* 8.3 (1994), pp. 298–299.

[Ber5] T. Berners-Lee. *star deployment scheme for Open Data*. https://www.w3.org/2011/gld/wiki/5_Star_Linked_Data. 5.

[Ber92] T. J. Berners-Lee. "The world-wide web". In: *Computer networks and ISDN systems* 25.4-5 (1992), pp. 454–459.

[BFM+98] T. Berners-Lee, R. Fielding, L. Masinter, et al. *Uniform resource identifiers (URI): Generic syntax*. 1998.

[BHB11a] C. Bizer, T. Heath, and T. Berners-Lee. "Linked data: The story so far". In: *Semantic services, interoperability and web applications: emerging concepts*. IGI Global, 2011, pp. 205–227.

[BHB11b] C. Bizer, T. Heath, and T. Berners-Lee. "Linked data: The story so far". In: *Semantic services, interoperability and web applications: emerging concepts*. IGI Global, 2011, pp. 205–227.

[BHH+06] D. Battré, F. Heine, A. Höing, and O. Kao. "On triple dissemination, forward-chaining, and load balancing in DHT based RDF stores". In: *Databases, Information Systems, and Peer-to-Peer Computing*. Springer, 2006, pp. 343–354.

[BHL01] T. Berners-Lee, J. Hendler, and O. Lassila. "The semantic web". In: *Scientific american* 284.5 (2001), pp. 34–43.

[BK03] J. Broekstra and A. Kampman. "Serql: A second generation rdf query language". In: *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*. 2003, pp. 13–14.

[Bla] Blazegraph Team. *Blazegraph - PropertyPaths*. https://wiki.blazegraph.com/wiki/index.php/PropertyPaths.

[BNT+08] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. "Bio2RDF: towards a mashup to build bioinformatics knowledge systems". In: *Journal of biomedical informatics* 41.5 (2008), pp. 706–716.

[BRB+14] W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, and S. Schlobach. "LOD laundromat: a uniform way of publishing other people's dirty data". In: *International Semantic Web Conference*. Springer. 2014, pp. 213–228.

[BWY17] Y. Bai, C. Wang, and X. Ying. "Para-G: Path pattern query processing on large graphs". In: *World Wide Web* 20.3 (2017), pp. 515–541.

[CB14] O. Curé and G. Blin. *RDF database systems: triples storage and SPARQL query processing*. Morgan Kaufmann, 2014.

[CBF+16] C. E. Cook, M. T. Bergman, R. D. Finn, G. Cochrane, E. Birney, and R. Apweiler. "The European Bioinformatics Institute in 2016: data growth and integration". In: *Nucleic acids research* 44.D1 (2016), pp. D20–D26.

[CD+99] J. Clark, S. DeRose, et al. *XML path language (XPath) version 1.0*. 1999.

[CDL+00] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. "Containment of conjunctive regular path queries with inverse". In: *KR 2000* (2000), pp. 176–185.

[CDL+03] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. "Reasoning on regular path queries". In: *ACM SIGMOD Record* 32.4 (2003), pp. 83–92.

[CEF+13] P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. F. Sequeda, and M. Wylot. "NoSQL databases for RDF: an empirical evaluation". In: *International Semantic Web Conference*. Springer. 2013, pp. 310–325.

[CEG+11] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaıda. *PSPARQL query containment*. https://hal.inria.fr/inria-00598819/. 2011.

[CF04] M. Cai and M. Frank. "RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network". In: *Proceedings of the 13th international conference on World Wide Web*. 2004, pp. 650–657.

[CFK+12] G. Cong, W. Fan, A. Kementsietsidis, J. Li, and X. Liu. "Partial evaluation for distributed XPath query processing and beyond". In: *ACM Transactions on Database Systems (TODS)* 37.4 (2012), pp. 1–43.

[CFK07] G. Cong, W. Fan, and A. Kementsietsidis. "Distributed query evaluation with performance guarantees". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 2007, pp. 509–520.

[CM90] M. P. Consens and A. O. Mendelzon. "GraphLog: a visual formalism for real life recursion". In: *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1990, pp. 404–416.

[CMW87] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. "A graphical query language supporting recursion". In: *ACM SIGMOD Record* 16.3 (1987), pp. 323–330.

[DAA05] F. Daniela, L. Alon, and M. Alberto. "Database techniques for the world wide web". In: *Sigmod record* 32.3 (2005), pp. 59–74.

[DE16] A. Davoust and B. Esfandiari. "Processing regular path queries on arbitrarily distributed data". In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2016, pp. 844–861.

[DG10] J. Dean and S. Ghemawat. "MapReduce: a flexible data processing tool". In: *Communications of the ACM* 53.1 (2010), pp. 72–77.

[DGH+14] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. "Knowledge vault: A web-scale approach to probabilistic knowledge fusion". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2014, pp. 601–610.

[Die12] R. Diestel. *Graph theory: Springer graduate text gtm 173*. Vol. 173. Reinhard Diestel, 2012.

[DuC11] B. DuCharme. *Learning SPARQL, Chapter 2 pp 19-44, Chapter 3 pp 45-100*. 2011.

[DVM16a] L. De Vocht, R. Verborgh, and E. Mannens. "Using Triple Pattern Fragments to Enable Streaming of Top-k Shortest Paths via the Web". In: *Semantic Web Evaluation Challenge*. Springer. 2016, pp. 228–240.

[DVM16b] L. De Vocht, R. Verborgh, and E. Mannens. "Using Triple Pattern Fragments to Enable Streaming of Top-k Shortest Paths via the Web". In: *Semantic Web Evaluation Challenge*. Springer. 2016, pp. 228–240.

[ELM+16] I. Ermilov, J. Lehmann, M. Martin, and S. Auer. "LODStats: The data web census dataset". In: *International Semantic Web Conference*. Springer. 2016, pp. 38–46.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext transfer protocol–HTTP/1.1*. 1999.

[FWC+13] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres. "SPARQL 1.1 Protocol". In: *Recommendation, W3C, March* (2013).

[FWW12] W. Fan, X. Wang, and Y. Wu. "Performance guarantees for distributed reachability queries". In: *arXiv preprint arXiv:1208.0091* (2012).

[GHS14] L. Galárraga, K. Hose, and R. Schenkel. "Partout: a distributed engine for efficient RDF processing". In: *Proceedings of the 23rd International Conference on World Wide Web*. 2014, pp. 267–268.

[Gir] Giraph Team. *Apache Giraph*. http://giraph.apache.org.

[GN11] A. Gubichev and T. Neumann. "Path Query Processing on Very Large RDF Graphs." In: *WebDB*. Citeseer. 2011.

[GS11a] O. Görlitz and S. Staab. "Splendid: Sparql endpoint federation exploiting void descriptions". In: *Proceedings of the Second International Conference on Consuming Linked Data-Volume 782*. CEUR-WS. org. 2011, pp. 13–24.

[GS11b] O. Görlitz and S. Staab. "Splendid: Sparql endpoint federation exploiting void descriptions". In: *Proceedings of the Second International Conference on Consuming Linked Data-Volume 782*. CEUR-WS. org. 2011, pp. 13–24.

[GSM+14] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. "TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 289–300.

[HAR11] J. Huang, D. J. Abadi, and K. Ren. "Scalable SPARQL querying of large RDF graphs". In: *Proceedings of the VLDB Endowment* 4.11 (2011), pp. 1123–1134.

[Har16] O. Hartig. *Querying a web of linked data: foundations and query execution*. Vol. 24. Ios Press, 2016.

[HB11] T. Heath and C. Bizer. "Linked data: Evolving the web into a global data space". In: *Synthesis lectures on the semantic web: theory and technology* 1.1 (2011), pp. 1–136.

[HBE+04] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. "A comparison of RDF query languages". In: *International Semantic Web Conference*. Springer. 2004, pp. 502–517.

[HHK05] F. Heine, M. Hovestadt, and O. Kao. "Processing complex RDF queries over P2P networks". In: *Proceedings of the 2005 ACM workshop on Information retrieval in peer-to-peer networks*. 2005, pp. 41–48.

[HKK+10] M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. "Data intensive query processing for large RDF graphs using cloud computing tools". In: *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE. 2010, pp. 1–10.

[HMM+11] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. "Heuristics-based query processing for large RDF graphs using cloud computing". In: *IEEE Transactions on Knowledge and Data Engineering* 23.9 (2011), pp. 1312–1327.

[HMR16] B. E. Huang, W. Mulyasasmita, and G. Rajagopal. "The path from big data to precision medicine". In: *Expert Review of Precision Medicine and Drug Development* 1.2 (2016), pp. 129–143.

[HMZ+17] A. Hasnain, Q. Mehmood, S. S. e Zainab, M. Saleem, C. Warren, D. Zehra, S. Decker, and D. Rebholz-Schuhmann. "BioFed: federated query processing over life sciences linked open data". In: *Journal of biomedical semantics* 8.1 (2017), p. 13.

[Hog11] A. Hogan. "Exploiting RDFS and OWL for Integrating Heterogeneous, Large-Scale, Linked Data Corpora". In: *Unpublished doctoral dissertation, National University of Ireland, Galway.(http://sw. deri. org/~ aidanh/docs/thesis/thesis-one-sided. pdf, retrieved October 2013)* (2011).

[HP17] O. Hartig and G. Pirrò. "SPARQL with Property Paths on the Web". In: *Semantic Web* 8.6 (2017), pp. 773–795.

[HQD15] W. Hu, H. Qiu, and M. Dumontier. "Link analysis of life science linked data". In: *International Semantic Web Conference*. Springer. 2015, pp. 446–462.

[HQI+16] Z. Hassan, M. A. Qadir, M. A. Islam, U. Shahzad, and N. Akhter. "Modified MinG algorithm to find top-k shortest paths from large RDF graphs". In: *Semantic Web Evaluation Challenge*. Springer. 2016, pp. 213–227.

[HSJ+16a] S. Hertling, M. Schröder, C. Jilek, and A. Dengel. "Top-k shortest paths in directed labeled multigraphs". In: *Semantic Web Evaluation Challenge*. Springer. 2016, pp. 200–212.

[HSJ+16b] S. Hertling, M. Schröder, C. Jilek, and A. Dengel. "Top-k shortest paths in directed labeled multigraphs". In: *Semantic Web Evaluation Challenge*. Springer. 2016, pp. 200–212.

[HSP13] S. Harris, A. Seaborne, and E. Prud'hommeaux. "SPARQL 1.1 query language". In: *W3C recommendation* 21.10 (2013), p. 778.

[JCB11] A. Jentzsch, R. Cyganiak, and C. Bizer. "State of the LOD Cloud". In: *Public Webpage* (2011).

[Jena] Jena-Fuseki-Team Team. *Apache Jena*. https://jena.apache.org/documentation/fuseki2/.

[Jenb] A. Jena. *Apache Jena TDB* -. https://jena.apache.org/documentation/tdb/.

[Jon96] N. D. Jones. "An introduction to partial evaluation". In: *ACM Computing Surveys (CSUR)* 28.3 (1996), pp. 480–503.

[KAC+02] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. "RQL: a declarative query language for RDF". In: *Proceedings of the 11th international conference on World Wide Web*. 2002, pp. 592–603.

[KJo7] K. J. Kochut and M. Janik. "SPARQLeR: Extended SPARQL for semantic association discovery". In: *European Semantic Web Conference*. Springer. 2007, pp. 145–159.

[KK95] G. Karypis and V. Kumar. "Analysis of multilevel graph partitioning". In: *Supercomputing'95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*. IEEE. 1995, pp. 29–29.

[KKH+18] S. Kawashima, T. Katayama, H. Hatanaka, T. Kushida, and T. Takagi. "NBDC RDF portal: a comprehensive repository for semantic data in life sciences". In: *Database* 2018 (2018).

[KKT+12] V. Khadilkar, M. Kantarcioglu, B. Thuraisingham, and P. Castagna. "Jena-HBase: A distributed, scalable and efficient RDF triple store". In: *Proceedings of the 11th International Semantic Web Conference Posters & Demonstrations Track, ISWC-PD*. Vol. 12. Citeseer. 2012, pp. 85–88.

[Kle51] S. C. Kleene. *Representation of events in nerve nets and finite automata*. Tech. rep. RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[Kly04] G. Klyne. *Resource description framework (RDF): Concepts and abstract syntax*. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. 2004.

[KM15] Z. Kaoudi and I. Manolescu. "RDF in the clouds: a survey". In: *The VLDB Journal* 24.1 (2015), pp. 67–91.

[KMA+03] G. Karvounarakis, A. Magganaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. "Querying the semantic web with RQL". In: *Computer networks* 42.5 (2003), pp. 617–640.

[Kos00] D. Kossmann. "The state of the art in distributed query processing". In: *ACM Computing Surveys (CSUR)* 32.4 (2000), pp. 422–469.

[KRU15] E. V. Kostylev, J. L. Reutter, and M. Ugarte. "CONSTRUCT queries in SPARQL". In: *18th International Conference on Database Theory (ICDT 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[KZJ+19] Y. Khan, A. Zimmermann, A. Jha, V. Gadepally, M. D'Aquin, and R. Sahay. "One size does not fit all: Querying web polystores". In: *Ieee Access* 7 (2019), pp. 9598–9617.

[Lin] Linked Open Data Cloud Team. *LOD Cloud* -. https://lod-cloud.net/.

[LKM+11] S. Lynden, I. Kojima, A. Matono, and Y. Tanimura. "Aderis: An adaptive query processor for joining federated sparql endpoints". In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2011, pp. 808–817.

[LL13] K. Lee and L. Liu. "Scaling queries over big RDF graphs with semantic hash partitioning". In: *Proceedings of the VLDB Endowment* 6.14 (2013), pp. 1894–1905.

[LLT+13] K. Lee, L. Liu, Y. Tang, Q. Zhang, and Y. Zhou. "Efficient and customizable data partitioning framework for distributed big RDF data processing in the cloud". In: *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE. 2013, pp. 327–334.

[LOÖ+14] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. "Distributed data management using MapReduce". In: *ACM Computing Surveys (CSUR)* 46.3 (2014), pp. 1–42.

[MAB+10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 135–146.

[MAF12] M. A. Martínez-Prieto, M. Arias, and J. D. Fernández. "Exchange and Consumption of Huge RDF Data". In: *The Semantic Web: Research and Applications*. Springer, 2012, pp. 437–452.

[MB12] H. Mühleisen and C. Bizer. "Web Data Commons-Extracting Structured Data from Two Large Web Corpora." In: *LDOW* 937 (2012), pp. 133–145.

[MJR+18] Q. Mehmood, A. Jha, D. Rebholz-Schuhmann, and R. Sahay. "FedS: Towards traversing federated rdf graphs". In: *International Conference on Big Data Analytics and Knowledge Discovery*. Springer. 2018, pp. 34–45.

[MLA+11] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. "DBpedia SPARQL benchmark–performance assessment with real queries on real data". In: *International semantic web conference*. Springer. 2011, pp. 454–469.

[Møl11] A. Møller. *Finite-state automata and regular expressions for Java*. https://www.brics.dk/automaton/. 2011.

[MP12] P. Mika and T. Potter. "Metadata Statistics for a Large Web Corpus." In: *LDOW* 937 (2012).

[MS00] D. Miller and D. Slater. *Internet*. Berg Publishers, 2000.

[MSM+15] G. Montoya, H. Skaf-Molli, P. Molli, and M.-E. Vidal. "Federated SPARQL queries processing with replicated fragments". In: *International Semantic Web Conference*. Springer. 2015, pp. 36–51.

[MSR02] L. Miller, A. Seaborne, and A. Reggiori. "Three implementations of SquishQL, a simple RDF query language". In: *International Semantic Web Conference*. Springer. 2002, pp. 423–435.

[MSS+19] Q. Mehmood, M. Saleem, R. Sahay, A.-C. N. Ngomo, and M. D'Aquin. "QPPDs: Querying Property Paths Over Distributed RDF Datasets". In: *IEEE Access* 7 (2019), pp. 101031–101045.

[MW95] A. O. Mendelzon and P. T. Wood. "Finding regular simple paths in graph databases". In: *SIAM Journal on Computing* 24.6 (1995), pp. 1235–1258.

[MYL10] J. Myung, J. Yeon, and S.-g. Lee. "SPARQL basic graph pattern processing with iterative MapReduce". In: *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*. 2010, pp. 1–6.

[Neo] Neo4j Team. *Neo4j -*. https://neo4j.com.

[NS16] M. Nolé and C. Sartiani. "Regular path queries on massive graphs". In: *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. 2016, pp. 1–12.

[NSW+09] N. F. Noy, N. H. Shah, P. L. Whetzel, B. Dai, M. Dorf, N. Griffith, C. Jonquet, D. L. Rubin, M.-A. Storey, C. G. Chute, et al. "BioPortal: ontologies and integrated data resources at the click of a mouse". In: *Nucleic acids research* 37.suppl_2 (2009), W170–W173.

[NW10] T. Neumann and G. Weikum. "The RDF-3X engine for scalable management of RDF data". In: *The VLDB Journal* 19.1 (2010), pp. 91–113.

[NWS+04] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Löser. "Super-peer-based routing strategies for RDF-based peer-to-peer networks". In: *Journal of Web Semantics* 1.2 (2004), pp. 177–186.

[Ont] Ontotext Team. *Ontotext LOD -*. https://bit.ly/2uSRoX4.

[OO02] M. Olson and U. Ogbuji. *Versa: Path-based RDF query language, 2002*. 2002.

[ORS+08] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. "Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 1099–1110.

[PAG10] J. Pérez, M. Arenas, and C. Gutierrez. "nSPARQL: A navigational language for RDF". In: *Journal of Web Semantics* 8.4 (2010), pp. 255–270.

[PB17] T. Padiya and M. Bhise. "DWAHP: workload aware hybrid partitioning and distribution of RDF data". In: *Proceedings of the 21st International Database Engineering & Applications Symposium*. 2017, pp. 235–241.

[PKT+12] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. "H2RDF: adaptive query processing on RDF data in the cloud." In: *Proceedings of the 21st International Conference on World Wide Web*. 2012, pp. 397–400.

[Prz17] M. Przyjaciel-Zablocki. "Distributed Processing of Navigational Query Languages for RDF". PhD thesis. Albert-Ludwigs-Universität Freiburg, 2017.

[PS+17] E. Prud'hommeaux, A. Seaborne, et al. *SPARQL query language for RDF. W3C Recommendation (2008)*. 2017.

[PSH+11] M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung, and G. Lausen. "Rdfpath: path query processing on large RDF graphs with mapreduce". In: *Extended Semantic Web Conference*. Springer. 2011, pp. 50–64.

[PTK+14] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. "H2RDF+ an efficient data management system for big RDF graphs". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 909–912.

[QL08a] B. Quilitz and U. Leser. "Querying distributed RDF data sources with SPARQL". In: *European semantic web conference*. Springer. 2008, pp. 524–538.

[QL08b] B. Quilitz and U. Leser. "Querying distributed RDF data sources with SPARQL". In: *European semantic web conference*. Springer. 2008, pp. 524–538.

[Rod15] M. A. Rodriguez. "The gremlin graph traversal machine and language (invited talk)". In: *Proceedings of the 15th Symposium on Database Programming Languages*. 2015, pp. 1–10.

[RS10] K. Rohloff and R. E. Schantz. "High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store". In: *Programming support innovations for emerging distributed applications*. 2010, pp. 1–5.

[SÇ18] M. Stonebraker and U. Çetintemel. "" One size fits all" an idea whose time has come and gone". In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2018, pp. 441–462.

[Sea] A. Seaborne. *SPARQL1.1 - PropertyPaths*. https://www.w3.org/TR/sparql11-property-paths/.

[Sea04] A. Seaborne. *Rdql-a query language for rdf*. http://www.w3.org/Submission/RDQL/. 2004.

[SEH+13] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel. "Horton+ a distributed system for processing declarative reachability queries over partitioned graphs". In: *Proceedings of the VLDB Endowment* 6.14 (2013), pp. 1918–1929.

[SHH+11] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. "FedX: a federation layer for distributed query processing on linked open data". In: *Extended Semantic Web Conference*. Springer. 2011, pp. 481–486.

[Sin12] A. Singhal. "Introducing the knowledge graph: things, not strings". In: *Official google blog* 16 (2012).

[Sir] E. Sirin. *Stardog - A path of our own*. https://www.stardog.com/blog/a-path-of-our-own/.

[SKH+16] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo. "A fine-grained evaluation of SPARQL endpoint federation systems". In: *Semantic Web* 7.5 (2016), pp. 493–518.

[SKR+10] K. Shvachko, H. Kuang, S. Radia, R. Chansler, et al. "The hadoop distributed file system." In: *MSST*. Vol. 10. 2010, pp. 1–10.

[SKW07] F. M. Suchanek, G. Kasneci, and G. Weikum. "Yago: a core of semantic knowledge". In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 697–706.

[SM20] L. H. Z. Santana and R. d. S. Mello. "An analysis of mapping strategies for storing rdf data into nosql databases". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 2020, pp. 386–392.

[SMU+17] V. Savenkov, Q. Mehmood, J. Umbrich, and A. Polleres. "Counting to k or how SPARQL1. 1 property paths can be extended to top-k path queries". In: *Proceedings of the 13th International Conference on Semantic Systems*. ACM. 2017, pp. 97–103.

[SN14a] M. Saleem and A.-C. N. Ngomo. "Hibiscus: Hypergraph-based source selection for sparql endpoint federation". In: *European semantic web conference*. Springer. 2014, pp. 176–191.

[SN14b] M. Saleem and A.-C. N. Ngomo. "Hibiscus: Hypergraph-based source selection for sparql endpoint federation". In: *European semantic web conference*. Springer. 2014, pp. 176–191.

[SN14c] M. Saleem and A.-C. N. Ngomo. "Hibiscus: Hypergraph-based source selection for sparql endpoint federation". In: *European semantic web conference*. Springer. 2014, pp. 176–191.

[SPH+11] A. Schätzle, M. Przyjaciel-Zablocki, T. Hornung, and G. Lausen. *Pigsparql: Übersetzung von sparql nach pig latin*. Gesellschaft für Informatik eV, 2011.

[SR13a] R. Simon and S. Roychowdhury. "Implementing personalized cancer genomics in clinical trials". In: *Nature reviews Drug discovery* 12.5 (2013), pp. 358–369.

[SR13b] R. Simon and S. Roychowdhury. "Implementing personalized cancer genomics in clinical trials". In: *Nature reviews Drug discovery* 12.5 (2013), pp. 358–369.

[SW13] S. Salihoglu and J. Widom. "Gps: A graph processing system". In: *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. 2013, pp. 1–12.

[SWL13] B. Shao, H. Wang, and Y. Li. "Trinity: A distributed graph engine on a memory cloud". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 505–516.

[Teaa] A. J. Team. *ARQ Property Paths* -. https://jena.apache.org/documentation/query/property_paths.html.

[Teab] O. S. D. Team. *Virtuoso Open Source Edition* -. http://vos.openlinksw.com/owiki/wiki/VOS.

[Teac] O. S. D. Team. *Virtuoso Property Paths* -. http://vos.openlinksw.com/owiki/wiki/VOS/VirtTipsAndTricksSPARQL11PropertyPaths.

[UHP+15] J. Umbrich, A. Hogan, A. Polleres, and S. Decker. "Link traversal querying for a diverse web of data". In: *Semantic Web* 6.6 (2015), pp. 585–624.

[Unm89] J. D. Unman. "Principles of database and knowledge-base systems". In: *Computer Science Press* (1989).

[Val90] L. G. Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[VCM08] K. Vipul, B. Christoph, and M. Matthew. "The Semantic Web-Semantics for Data and Services on the Web". In: *Berlin and Heidelberg* (2008).

[VSN+18] A. Valdestilhas, T. Soru, M. Nentwig, E. Marx, M. Saleem, and A.-C. N. Ngomo. "Where is my URI?" In: *European Semantic Web Conference*. Springer. 2018, pp. 671–681.

[WDS+12] D. J. Wild, Y. Ding, A. P. Sheth, L. Harland, E. M. Gifford, and M. S. Lajiness. "Systems chemical biology and the Semantic Web: what they mean for the future of drug discovery research". In: *Drug discovery today* 17.9-10 (2012), pp. 469–474.

[Wik] Wikipedia. *Distributed databases* -. https://en.wikipedia.org/wiki/Distributed_database.

[WWX+19] X. Wang, S. Wang, Y. Xin, Y. Yang, J. Li, and X. Wang. "Distributed Pregel-based provenance-aware regular path query processing on RDF knowledge graphs". In: *World Wide Web* (2019), pp. 1–32.

[WWZ16a] X. Wang, J. Wang, and X. Zhang. "Efficient distributed regular path queries on RDF graphs using partial evaluation". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM. 2016, pp. 1933–1936.

[WWZ16b] X. Wang, J. Wang, and X. Zhang. "Efficient distributed regular path queries on RDF graphs using partial evaluation". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 2016, pp. 1933–1936.

[XWJ+18] Y. Xin, X. Wang, D. Jin, and S. Wang. "Distributed efficient provenance-aware regular path queries on large RDF graphs". In: *International Conference on Database Systems for Advanced Applications*. Springer. 2018, pp. 766–782.

[YC10a] J. X. Yu and J. Cheng. "Graph reachability queries: A survey". In: *Managing and Mining Graph Data*. Springer, 2010, pp. 181–215.

[YC10b] J. X. Yu and J. Cheng. "Graph reachability queries: A survey". In: *Managing and Mining Graph Data*. Springer, 2010, pp. 181–215.

[ZCF+10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[ZYW+13] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. "A distributed graph engine for web scale RDF data". In: *Proceedings of the VLDB Endowment* 6.4 (2013), pp. 265–276.