Querying Online Decentralized Knowledge Graphs

Julien Aimonier-Davat

THÈSE DE DOCTORAT

NANTES UNIVERSITÉ

ÉCOLE DOCTORALE N° 641
Mathématiques et Sciences et Technologies du numérique de l'Information et de la Communication
Spécialité : *Informatique*

Par
Julien AIMONIER-DAVAT

Querying Online Decentralized Knowledge Graphs

Thèse présentée et soutenue à Nantes, le 11 décembre 2023
Unité de recherche : Laboratoire des sciences du numérique à Nantes (LS2N)

Rapporteurs avant soutenance :

Frédérique LAFOREST	Professeure des universités, INSA Lyon
Mathieu D'AQUIN	Professeur des universités, Université de Lorraine, Nancy

Composition du Jury :

Président :	Arnaud SOULET	Professeur des Universités, Université de Tours
Examinateurs :	Frédérique LAFOREST	Professeure des Universités, INSA Lyon
	Mounira HARAZALLAH	Maître de Conférences HDR, Nantes Université
	Mathieu D'AQUIN	Professeur des Universités, Université de Lorraine, Nancy
	Luis CALARRAGE	Chargé de Recherches, INRIA Rennes
Dir. de thèse :	Hala SKAF-MOLLI	Professeure des Universités, Nantes Université
Co-dir. de thèse :	Pascal MOLLI	Professeur des Universités, Nantes Université

TABLE OF CONTENTS

Part I

Introduction and Background

Chapter 1

Introduction

Contents

1.1 Knowledge Graphs

The idea of the Semantic Web started with Tim Berners-Lee [29], the person who created the World Wide Web, back in the 1990s. The main goal of the Semantic Web is to make the regular Web easy for computers to understand, helping institutions, companies, and people share knowledge. The Web can then be seen as a huge "knowledge database" that any computer can ask questions to, in order to perform sophisticated tasks for users [29]. To make this vision come true, the Semantic Web community came up with many standards, like the Resource Description Framework [91] (RDF) to represent knowledge, and SPARQL [61], which is a language for querying knowledge.

Thanks to RDF, Knowledge Graphs (KGs) can be defined as a collection of facts. For instance, the DBpedia knowledge graph [82] is created by extracting information from Wikipedia and presenting it in RDF format. Figure 1.1 depicts a fragment of DBpedia that provides details about Barack Obama, the former President of the United

```
@prefix dbo: <http://dbpedia.org/ontology/>                          .
@prefix dbr: <http://dbpedia.org/resource/>                          .
@prefix dbp: <http://dbpedia.org/property/>                          .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>           .
@prefix owl: <http://http://www.w3.org/2002/07/owl#>                 .

dbr:Barack_Obama    rdf:type    dbo:Person                           ;
    dbp:birthName   "Barack Hussein Obama II"                        ;
    dbp:title       dbr:President_of_the_United_States               ;
    dbp:before      dbr:George_W._Bush                               ;
    dbo:author      dbr:Dreams_from_My_Father, dbr:A_Promised_Land    .

dbr:Barack_Obama    owl:sameAs  <http://viaf.org/viaf/52010985>       ,
    <http://d-nb.info/gnd/132522136>                                 ,
    <http://data.europa.eu/euodp/jrc-names/Barack_Obama>             ,
    <http://musicbrainz.org/artist/...7abdc144f1d5>                  ,
    <http://www.wikidata.org/entity/Q76>                             ,
    <http://data.bibliotheken.nl/id/thes/p287770850>                 .
```

Figure 1.1 – Turtle representation of RDF triples, extracted from DBpedia, about the entity Barack Obama.

States. In this representation, information about President Obama is structured as a series of triples $(Subject, Property, Value)$, with each triple asserting a specific fact. The first triple indicates that Barack Obama is classified as a Person. The second triple reveals his birth name. Subsequent triples inform us that he served as the President of the United States after George W. Bush and that he authored books such as "Dream from My Father" and "A Promised Land".

The DBpedia knowledge graph contains nearly 9.5 billion RDF triples, capturing a significant portion of Wikipedia's knowledge across various languages. If a user wishes to know more about Barack Obama, she can dereference the URI `https://dbpedia.org/resource/Barack_Obama`, which returns all the triples DBpedia has regarding this entity. This information can be presented in HTML for human readability or in RDF format for computer processing.

A simple way of querying knowledge graphs is to use the SPARQL query language. If we want to know who became the U.S. president after George W. Bush, we can write the SPARQL query shown in Figure 1.2. This query involves joining the result of two

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

SELECT ?s WHERE {
    ?s  dbp:title   dbr:President_of_the_United_States  .
    ?s  dbp:before  dbr:George_W._Bush                  .
}
```

Figure 1.2 – A simple SPARQL query over the DBpedia knowledge graph. Retrieves the U.S. president that is the successor of George W. Bush.

triple patterns. The first pattern searches for U.S. presidents, while the second searches for successors of George W. Bush. In this context, $?s$ is the entity we are looking for, i.e., an entity that is defined as both a president and the successor of George W. Bush. To execute this query, we can use the public SPARQL endpoint of DBpedia, which can be accessed at `https://dbpedia.org/sparql#`. A SPARQL endpoint is a Web server designed to execute SPARQL queries directly on knowledge graphs. The main advantage of SPARQL endpoints is to provide a standardized interface [48] for querying RDF data. Executing our query on DBpedia, we get the following result: `https://dbpedia.org/resource/Barack_Obama`.

Following the W3C recommendations, the Semantic Web community has made RDF data available online through SPARQL endpoints. The most striking examples of these online knowledge graphs are DBpedia and Wikidata, which enable users to answer complex queries. These queries can range from identifying the descendants of Bach to discovering streets in France named after women, or even finding information about bridges spanning the rivers of Leipzig [1].

1.2 Decentralized Knowledge Graphs

The real power of knowledge graphs lies in their ability to connect with one another. To illustrate, the last triples in Figure 1.1 use the *sameAs* property to link `https://dbpedia.org/entity/Barack_Obama` to other resources that describe the same real-

1. `https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`

world entity in other knowledge graphs. Each of these resources allows us to learn more about Barack Obama. For example, the New York Times knowledge graph gives us news about the former president of the U.S., while the MusicBrainz knowledge graph provides us with information on his discography. By linking its resources with other resources, DBpedia follows the Linked Data principles proposed by Tim Berners-Lee [28]:

- Use URI to identify things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide information using standards.
- Include links to other URIs so that they can discover more things.

The Linked Data principles aim to help the Semantic Web community publish, share, and interlink data on the Web. Following these principles, data providers published hundreds of knowledge graphs over the Web [30, 114]. All these interlinked knowledge graphs create a large decentralized knowledge graph that everyone can interact with and contribute to. This graph is named the Linked Open Data Cloud (LOD Cloud). The current state of the LOD Cloud is illustrated in Figure 1.3. As of September 2023, it contains 1314 datasets, 16308 links, and covers many topics. For example, we can find knowledge graphs focusing on institutional data (e.g., city or government public data), linguistic data (e.g., dictionaries, corpora or typography lists), life sciences (e.g., biological databases), or general knowledge such as encyclopedias.

Using the SPARQL query language, any user can execute a SPARQL query over multiple SPARQL endpoints. For example, the query depicted in Figure 1.4a retrieves information about Barack Obama from both DBpedia and Wikidata. We use the SERVICE clauses introduced in SPARQL 1.1 to indicate on which SPARQL endpoints each part of the query should be executed. The first clause is executed on Wikidata to retrieve all the information corresponding to the resource `https://www.wikidata.org/entity/Q76`, i.e., Barack Obama on Wikidata. The second clause is executed on DBpedia and uses the *sameAs* link in Figure 1.1 to find the resource `https://dbpedia.org/entity/Barack_Obama` on DBpedia and retrieve DBpedia information about Barack Obama. We call queries like the one described in Figure 1.4a *SERVICE queries*.

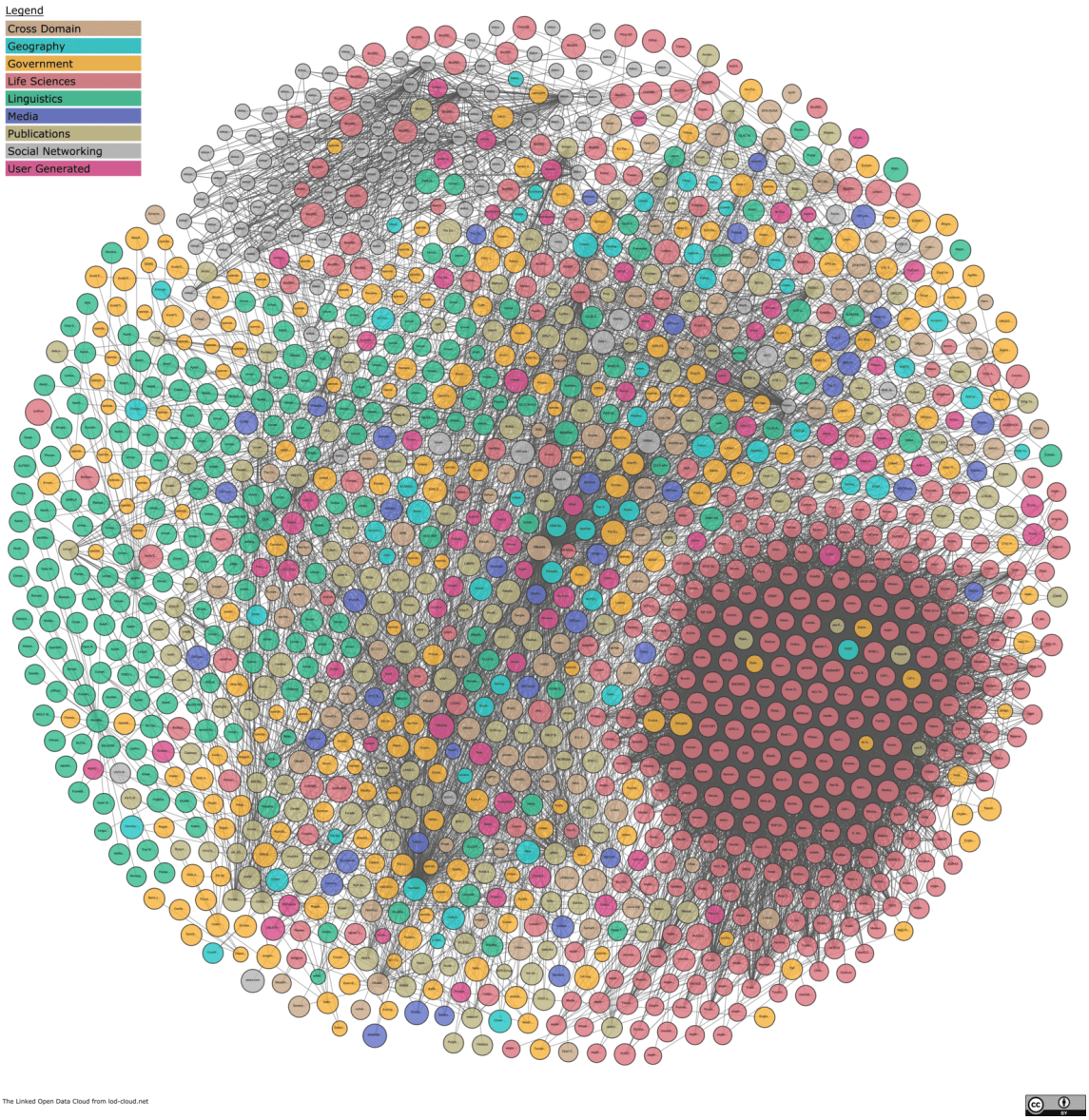


Figure 1.3 – The LOD Cloud diagram, as of September 2023.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX wd: <http://www.wikidata.org/entity/>

SELECT ?p ?o WHERE {
  {
    SERVICE <https://query.wikidata.org> {
      wd:Q76 ?p ?o .
    }
  } UNION {
    SERVICE <https://dbpedia.org/sparql> {
      ?s owl:sameAs wd:Q76 . ?s ?p ?o .
    }
  }
}
```

(a) Service query

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX wd: <http://www.wikidata.org/entity/>

SELECT ?p ?o WHERE {
  {
    wd:Q76 ?p ?o .
  } UNION {
    ?s owl:sameAs wd:Q76 . ?s ?p ?o .
  }
}
```

(b) Federated query

Figure 1.4 – SPARQL query that retrieves all information about Barack Obama on Wikidata and DBpedia.

1.3 SPARQL Federation Engines

Querying multiple knowledge graphs using SERVICE queries is cumbersome when the number of knowledge graphs increases. For example, to find all the information about Barack Obama on the LOD Cloud, we need to contact all SPARQL endpoints, i.e., to extend the query in Figure 1.4a with hundreds of SERVICE clauses, which is not realistic. A more convenient way to query the LOD Cloud should be to execute the query presented in Figure 1.4a without explicitly specifying the SPARQL endpoints to contact. Regular SPARQL queries that require multiple SPARQL endpoints to produce a complete answer are called *federated queries*.

Federated queries can be executed using federation engines such as FedX [118]. The input of a federation engine is a regular SPARQL query, and a list of SPARQL endpoints. It then virtually decomposes the federated query into a SERVICE query that can be executed on the relevant SPARQL endpoints. For example, when provided with the federated query depicted in Figure 1.4b and a list of two endpoints, namely DBpedia and Wikidata, a federation engine can generate the SERVICE query shown in Figure 1.4a.

Federation engines represent a significant approach within the realm of the Semantic Web for querying a decentralized Web of public SPARQL endpoints. This approach treats a decentralized knowledge graph as interconnected knowledge graphs following the Linked Data principles and accessible through public SPARQL endpoints. A

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

SELECT ?creativeWork ?fictionalWork WHERE {
    ?creativeWork wdt:P31/wdt:P279* wd:Q17537576 . # creative work
    ?creativeWork wdt:P144 ?fictionalWork .
    ?fictionalWork wdt:P136 wd:Q8253 . # fiction
}
```

Figure 1.5 – Query $Q1$. Retrieves creative works and the list of fictional works that inspired them on Wikidata.

collection of SPARQL endpoints is named a *federation*. Given a federated query and a federation, a federation engine is responsible for returning complete and correct results for the federated query across the federation.

1.4 Issues of SPARQL Federation Engines

Unfortunately, evaluating federated queries on a federation of online knowledge graphs raises two severe issues: (1) SPARQL endpoints do not ensure complete results (2) Federation engines do not scale on large federations.

Public SPARQL endpoints do not ensure complete results SPARQL federation engines assume that the underlying SPARQL endpoints will return complete results, which is not true in practice [92]. Public SPARQL query services must deal with unpredictable loads of arbitrary SPARQL queries. The challenge for public endpoints is to ensure that the service remains available despite significant variations in query arrival rates and resources required for query processing. Data providers often refer to this challenge as "maintaining a fair usage policy". To achieve a fair usage policy, most SPARQL endpoints enforce quotas. For example, the public SPARQL endpoint of Wikidata interrupts all queries that are still running after 60 seconds. Quotas ensure that server resources are shared fairly between users but prevent complete results.

To illustrate, Query $Q1$ in Figure 1.5 returns creative works inspired by fictional works. When we execute $Q1$ on the public SPARQL endpoint of Wikidata, the query times out and returns only partial results. Query $Q1$ is not an isolated case; numer-

ous queries on Wikidata face similar issues [33]. According to Wikidata query logs [2], thousands of queries time out each month. Unsurprisingly, long-running queries such as property path and aggregate queries are frequently encountered in these logs. So much so that property path queries are one of the main focuses of the latest Wikidata benchmark [17]. As knowledge graphs are continuously growing and SPARQL queries are becoming more and more complex [14, 101], we expect more and more queries to be interrupted by the fair use policy of public SPARQL endpoints.

Delivering partial results is a severe limitation as it prevents Semantic Web applications from relying on public SPARQL endpoints. Critical use cases are not achievable without the guarantee of complete results. For example, SPARQL endpoints should provide VoID descriptions [13] to enable a wide range of applications, ranging from data discovery to dataset cataloging and archiving. Unfortunately, in practice, many VoID descriptions are not available [70]. While VoID descriptions could be computed using SPARQL queries [70], fair use policies prevent VoID queries from returning complete results [70, 89]. The same problem arises when federation engines must compute summaries. If not provided, the summaries could be computed by federation engines themselves using SPARQL queries. However, HiBiSCuS [110] and CostFed [112] summaries cannot be obtained online because of public SPARQL endpoints fair use policy.

Undoubtedly, existing SPARQL endpoints are valuable. However, in their quest to provide fair services, they have made trade-offs, often sacrificing completeness. Because ensuring complete results is crucial for many use cases, the first challenge of this thesis is to provide *a fair SPARQL query service that ensures complete results*. Such a SPARQL query service embodies the concept of *fair query processing*.

Federation engines do not scale on large federations Assuming public SPARQL endpoints manage to provide complete results, scalability becomes a challenge for SPARQL federation engines when dealing with an increasing number of data sources [1, 42].

To illustrate this point, we refer to the FedShop benchmark [42], which simulates an e-commerce scenario with online shops and rating sites. Each shop or site is a knowledge graph connected to others through similar products. FedShop queries simulate users searching for products, reviews, or similar products across the federation. For example, Query $Q5$ in Figure 1.6a supposes that a user is looking for similar products to those she is interested in. We executed Query $Q5$ over two federations of 20 (10 shops

2. `https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en`

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?product ?localProductLabel WHERE {
    ?localProductXYZ owl:sameAs bsbm:Product43923   ;
        bsbm:productPropertyNumeric1 ?origProperty1 ;
        bsbm:productPropertyNumeric2 ?origProperty2 ;
        bsbm:productFeature ?localProdFeatureXYZ .

    ?localProdFeatureXYZ owl:sameAs ?prodFeature .
    ?localProdFeature owl:sameAs ?prodFeature .

    ?localProduct bsbm:productFeature ?localProdFeature ;
        bsbm:productPropertyNumeric1 ?simProperty1 ;
        bsbm:productPropertyNumeric2 ?simProperty2 ;
        rdfs:label ?localProductLabel ;
        owl:sameAs ?product .

    FILTER(bsbm:Product43923 != ?product)
    FILTER(?simProperty1 < (?origProperty1 + 20) && ?simProperty1 > (?origProperty1 - 20))
    FILTER(?simProperty2 < (?origProperty2 + 70) && ?simProperty2 > (?origProperty2 - 70))
}
```

(a) FedShop Query $Q5$. Returns similar products to Product43923.

	20 shops	200 shops
RSA	50ms	1.5 seconds
CostFed	3 seconds	$>$ 1 hour

(b) Execution time of Query $Q5$ on 20 and 200 shops using CostFed and hand-crafted (RSA) query execution plans.

Figure 1.6 – Execution time of FedShop Query $Q5$ on 20 and 200 shops using CostFed and hand-crafted (RSA) query execution plans.

and 10 rating sites) and 200 (100 shops and 100 rating sites) sources, and compared the performance of CostFed [112], i.e., the best-performing federation engine to date, with hand-crafted SERVICE queries. The SERVICE query for $Q5$ is provided by the Reference Source Assignment (RSA) in FedShop. Table 1.6b reports CostFed and RSA execution times for $Q5$ on both federations.

1. In terms of performance, the RSA outperforms CostFed on both federations. In the case of the 20-source federation, the RSA is roughly 60 times faster than CostFed, whereas across the larger 200-source federation, it exhibits a minimum speed advantage of 3600 times. Waiting more than 1 hour to get similar products is impossible in a dynamic scenario like an e-commerce application.
2. Regarding scalability, when we increase the federation size by a factor of 10, the

RSA execution time increases by a factor of 30. In contrast, the execution time of CostFed multiplies by a minimum of 120.

The results obtained from FedShop make it clear that current federation engines are limited to handling small federations, with typically less than 20 sources. In contrast, the LOD Cloud contains hundreds of knowledge graphs. Thankfully, the performance of RSA shows that federation engines can be drastically improved. As a result, the second challenge of this thesis is to *address the performance and scalability issues of existing federation engines*.

1.5 Research Questions and Contributions

In this thesis, we aim to answer the following research questions:

1. How to provide a fair SPARQL query service for online knowledge graphs that ensures complete results?
2. How to improve the scalability of SPARQL federation engines?

1.5.1 How to Provide a Fair SPARQL Query Service for Online Knowledge Graphs that Ensures Complete Results?

Public SPARQL endpoints have made a deliberate choice: sacrificing completeness for fairness. However, sacrificing completeness makes many use cases impossible. To solve this problem, many approaches have been proposed that achieve both completeness and fairness [4, 67, 134]. The key idea is to distribute the query processing load between the server and the client. The server only handles SPARQL operators that comply with fair query processing, while the client handles the remaining operators. However, these approaches introduce a significant overhead that impacts performance.

Recently, Web Preemption [92] introduced a new query execution model for public SPARQL servers. Web Preemption allows servers to suspend a running SPARQL query and resume it later. This Stop-and-Go mechanism offers many advantages: it ensures a fair allocation of server resources, guarantees complete results, and expands the server's capacity to support a broader range of operators compared to previous approaches. Consequently, Web Preemption leads to significant improvements in query execution performance for many classes of queries.

However, property path queries with transitive closures, as well as aggregate queries using COUNT-DISTINCT aggregation functions, remain poorly supported by Web Preemption [8, 11]. Improving query execution performance for these queries is imperative as they play a crucial role in numerous use cases. For example, property path queries with transitive closures enable users to search knowledge graphs for paths of arbitrary lengths, while the COUNT-DISTINCT aggregation function plays a pivotal role in computing a wide range of valuable statistics [51, 70, 89, 112].

Web Preemption set aside, the order in which the different query clauses are executed, commonly known as the *join order*, significantly impacts query execution performance [84]. Finding optimal join orders is of paramount importance for providing an efficient SPARQL query service. A suboptimal join order can transform a query that typically runs in seconds into one that takes days to complete.

To tackle these performance issues, we have two contributions. First, we extend the Web Preemption model to improve execution performance of property path and aggregate queries. Second, we propose a new optimizer to find better join orders for conjunctive SPARQL queries with filters and property paths.

1.5.2 How to Improve the Scalability of SPARQL Federation Engines?

The performance of SPARQL federation engines is closely tied to the source selection and query decomposition problem [39]. Given a set of SPARQL endpoints and a SPARQL query, the key question is: which parts of the query should be evaluated on which endpoints to get complete results while minimizing the number of sub-queries? Finding the minimum number of sub-queries means pushing as much computation as possible to the local SPARQL endpoints. With poor query decomposition, federation engines must retrieve more intermediate results, significantly degrading query execution performance [42].

To improve the scalability of federation engines, we propose FedUP, a new federation engine. Compared to existing federation engines, FedUP introduces a new result-aware source selection algorithm based on random walks. Moreover, FedUP outputs the source selection as a logical query plan built to capture relevant information for query optimizers.

1.6 Thesis Overview

This thesis is structured into four parts:

- Part I includes this introductory chapter and chapter 2, which introduces fundamental concepts related to the Resource Description Framework and the SPARQL query language.
- Part II constitutes the core of our contributions towards offering a fair SPARQL query service that ensures complete results and low performance overhead. This part is composed of five chapters. Chapter 4 delves into an exploration of state-of-the-art SPARQL query services that ensure completeness and fairness, along with their associated limitations. Chapters 5 and 6 present our extensions to the Web Preemption model, aimed at improving query execution performance for property path and COUNT-DISTINCT queries, respectively. Finally, in Chapter 7, we introduce a solution to optimize the join orders of SPARQL queries, especially in the presence of property paths.
- Part III focuses on our contributions to enhance the scalability of SPARQL federation engines. This part comprises three chapters. Chapter 8 serves as a brief introduction to the scalability issue of SPARQL federation engines. In Chapter 9, we provide a comprehensive review of existing federation engines. Finally, in Chapter 10, we elaborate on our proposal, FedUP, which is an innovative federation engine that generates more effective source assignments than those produced by existing solutions.
- Part IV concludes this thesis and outlines perspective works.

1.7 Publications

(1) **RAW-JENA: Approximate Query Processing for SPARQL Endpoints**, by Julien Aimonier-Davat, Minh-Hoang Dang, Pascal Molli, Brice Nédelec and Hala Skaf-Molli. In the International Semantic Web Conference, 2023.
Abstract: Sampling-based Approximate Query Processing (S-AQP) has many important use cases for RDF, including computing large-scale statistics, embeddings, join orderings, approximate aggregations, summaries, and exploratory queries. However, current SPARQL endpoints have no support for S-AQP, and many queries just time out on public SPARQL endpoints. In this demonstration, we present RAW-JENA: an extension of Apache Jena to support S-AQP for conjunctive SPARQL

queries relying on random walks. RAW-JENA delivers partial random results and cardinality estimates in a pay-as-you-go fashion.

(2) **FedShop: A Benchmark for Testing the Scalability of SPARQL Federation Engines**, by Minh-Hoang Dang, Julien Aimonier-Davat, Pascal Molli, Olaf Hartig, Hala Skaf-Molli and Yotlan Le Crom. In the International Semantic Web Conference, 2023.

Abstract: While several approaches to query a federation of SPARQL endpoints have been proposed in the literature, very little is known about the effectiveness of these approaches and the behavior of the resulting query engines for cases in which the number of federation members increases. The existing benchmarks that are typically used to evaluate SPARQL federation engines do not consider such a form of scalability. In this paper, we set out to close this knowledge gap by investigating the behavior of 4 state-of-the-art SPARQL federation engines using a novel benchmark designed for scalability experiments. Based on the benchmark, we show that scalability is a challenge for each of these engines, especially with respect to the effectiveness of their source selection and query decomposition approaches.

(3) **Join Ordering of SPARQL Property Path Queries**, by Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli, Minh-Hoang Dang and Brice Nédelec. In the European Semantic Web Conference, 2023.

Abstract: SPARQL property path queries provide a succinct way to write complex navigational queries over RDF knowledge graphs. However, their evaluation remains difficult as they may involve the execution of transitive closures. As a result, many property path queries just time-out when executed on public online RDF knowledge graphs. One solution to speed up their execution is to find optimal join orders. Although the join ordering problem has been extensively studied for traditional SPARQL queries, the presence of property path patterns biases existing approaches. In this paper we focus on $C2RPQ_{UF}$ queries (conjunctive SPARQL property path queries with UNION and FILTER), and we present a query optimizer that is able to capture the cost of $C2RPQ_{UF}$ queries using an appropriate cost model and a sampling-based cardinality estimator. On the latest Wikidata Query Benchmark, we empirically demonstrate that our approach finds significantly better join orders than Virtuoso and Blazegraph.

(4) **Online approximative SPARQL query processing for COUNT-DISTINCT queries with Web Preemption**, by Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli,

Arnaud Grall and Thomas Minier. In Semantic Web, 2022.
Abstract: Getting complete results when processing aggregate queries on public SPARQL endpoints is challenging, mainly due to the application of quotas. Although Web preemption supports processing of aggregate queries online, on preemptable SPARQL servers, data transfer is still very large when processing count-distinct aggregate queries. In this paper, it is shown that count-distinct aggregate queries can be approximated with low data transfer by extending the partial aggregation operator with HyperLogLog++ sketches. Experimental results demonstrate that the proposed approach outperforms existing approaches by orders of magnitude in terms of the amount of data transferred.

(5) **Processing SPARQL TOP-k Queries Online with Web Preemption**, by Julien Aimonier-Davat, Hala Skaf-Molli and Pascal Molli. In the 6th Workshop on Storing, Querying and Benchmarking the Web of Data, 2022.
Abstract: Processing top-k queries on public online SPARQL endpoints often runs into fair use policy quotas and does not complete. Indeed, existing endpoints mainly follow the traditional materialize-and-sort strategy. Although restricted SPARQL servers ensure the termination of top-k queries without quotas enforcement, they follow the materialize-and-sort approach, resulting in high data transfer and poor performance. In this paper, we propose to extend the Web preemption model with a preemptable partial top-k operator. This operator drastically reduces data transfer and significantly improves query execution time. Experimental results show a reduction in data transfer by a factor of 100 and a reduction of up to 39% in Wikidata query execution time.

(6) **Processing SPARQL Property Path Queries Online with Web Preemption**, by Julien Aimonier-Davat, Hala Skaf-Molli and Pascal Molli. In the European Semantic Web Conference, 2021.
Abstract: SPARQL property path queries provide a succinct way to write complex navigational queries over RDF knowledge graphs. However, evaluating these queries over online knowledge graphs such as DBpedia or Wikidata is often interrupted by quotas, returning no or partial results. To ensure complete results, property path queries are evaluated client-side. Smart clients decompose property path queries into subqueries for which complete results are ensured. The granularity of the decomposition depends on the expressivity of the server. Whatever the decomposition, it could generate a high number of subqueries, a large data trans-

fer, and finally deliver poor performance. In this paper, we extend a preemptable SPARQL server with a partial transitive closure operator (PTC) based on a depth limited search algorithm. We show that a smart client using the PTC operator can process SPARQL property path online and deliver complete results. Experimental results demonstrate that our approach outperforms existing smart client solutions regarding HTTP calls, data transfer, and query execution time.

(7) **SaGe-Path: Pay-as-you-go SPARQL Property Path Queries Processing using Web Preemption**, by Julien Aimonier-Davat, Hala Skaf-Molli and Pascal Molli. In the European Semantic Web Conference, 2021.
Abstract: SPARQL property path queries allow to write sophisticated navigational queries on knowledge graphs (KGs). However, evaluating these queries on online KGs are often interrupted by fair use policies, returning only partial results. SaGe-Path addresses this issue by relying on the Web preemption and the concept of Partial Transitive Closure (PTC). Under PTC, the graph exploration for SPARQL property path queries is limited to a predefined depth. When the depth limit is reached, frontier nodes are returned to the client. A PTC-client can then reuse frontier nodes to continue the exploration of the graph. In this way, SaGe-Path follows a pay-as-you-go approach to evaluate SPARQL property path queries. This demonstration shows how queries not complete on the public Wikidata SPARQL endpoint can be completed using SaGe-Path. An extended user-interface provides real-time visualization of all SaGe-Path internals, allowing one to understand the approach overheads and the effects of different parameters on performance. SaGe-Path demonstrates how complex SPARQL property path queries can be efficiently evaluated online with guaranteed complete results.

(8) **How to execute SPARQL property path queries online and get complete results?**, by Julien Aimonier-Davat, Hala Skaf-Molli and Pascal Molli. In the 4th Workshop on Storing, Querying and Benchmarking the Web of Data, 2020.
Abstract: SPARQL property path queries provide a concise way to write complex navigational queries over RDF knowledge graphs. However, the evaluation of these queries over online knowledge graphs such as DBpedia or Wikidata are often interrupted by quotas, returning no results or partial results. Decomposing SPARQL property path queries into triple pattern subqueries allow us to get complete results. However, such decomposition generates a high number of subqueries, a large data transfer, and finally delivers poor performances. In this paper, we propose an

algorithm able to decompose SPARQL property path queries into Basic Graph Pattern (BGP) subqueries. As BGP queries are guaranteed to terminate on preemptable SPARQL servers, property path queries always deliver complete results. Experimental results demonstrate that our approach outperforms existing approaches regarding HTTP calls, data transfer, and query execution time.

(9) **FedUP: A Pay-as-you-go Federated SPARQL Query Engine Powered by Random Walks**, by Julien Aimonier-Davat, Brice Nédelec, Pascal Molli, Hala Skaf-Molli and Minh-Hoang Dang. To be submitted to the TheWebConf 2024.
Abstract: Federated SPARQL query engines allow for the integration and querying of multiple SPARQL endpoints as if they were a single one. While federated query engines avoid moving RDF data, performance remains a critical issue deeply linked to minimizing the number of subqueries sent to endpoints. As this problem of source selection is NP-hard, existing federated query engines build suboptimal query plans that may significantly degrade performance. In this paper, we propose FedUP, a federated query engine that relies on random walks to approximate source selection in a pay-as-you-go fashion. By exploring federated graphs (or their summaries), random walks build join-aware logical plans by better capturing the relationships between sources, and reducing the number of subqueries sent to endpoints. As many results share identical combinations of endpoints, random walks quickly converge to an accurate source selection that returns complete and correct results. Experimental studies on federated benchmarks demonstrate that FedUP outperforms state-of-the-art federated query engines regarding source selection and query execution time.

Chapter 2

Background

Contents

This chapter introduces fundamental concepts related to the Resource Description Framework and the SPARQL query language. Our formalism is based on commonly used notations and definitions [79, 100, 115].

2.1 The Resource Description Framework

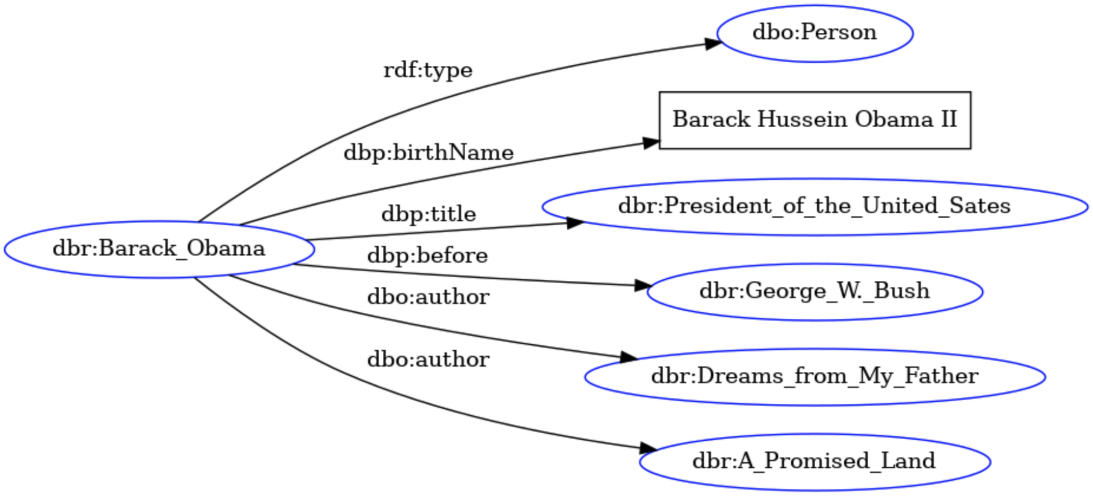
The Resource Description Framework [91] (RDF) is a standardized data model for describing resources and their relationships in a structured and machine-readable format. The building blocks of RDF are resources. A resource represents anything that can be identified, described, or referenced on the Web. It can be physical objects, abstract concepts, digital documents, or intangible entities. Each resource is identified using a Uniform Resource Identifier (URI). For example, `https://dbpedia.org/resource/Barack_Obama` is the URI of a resource about the entity Barack Obama. In RDF, data are represented as statements, known as RDF triples, each composed of a subject (the resource being described), a predicate (the relationship or attribute), and an object (another resource or a literal value). To illustrate, Figure 2.1a shows RDF triples that assert facts about the entity Barack Obama. A collection of RDF triples is called an RDF graph, and a set of RDF graphs is an RDF dataset. In a way, an RDF graph is a directed and labeled multi-graph, where an RDF triple is a directed edge whose predicate is the label, the subject is the source node, and the object is the target node. For example, Figure 2.1b shows the same data as Figure 2.1a but represented as a directed, labeled RDF graph.

More formally, using common notations [100], we define three disjoint sets I, L, B, for URIs, literals, and blank nodes, respectively. Then, we have the following definitions:

```
@prefix dbo: <http://dbpedia.org/ontology/>                          .
@prefix dbr: <http://dbpedia.org/resource/>                          .
@prefix dbp: <http://dbpedia.org/property/>                          .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>           .
@prefix owl: <http://http://www.w3.org/2002/07/owl#>                 .

dbr:Barack_Obama    rdf:type    dbo:Person                           ;
    dbp:birthName   "Barack Hussein Obama II"                        ;
    dbp:title       dbr:President_of_the_United_States               ;
    dbp:before      dbr:George_W._Bush                               ;
    dbo:author      dbr:Dreams_from_My_Father, dbr:A_Promised_Land    .
```

(a) Turtle representation of RDF triples about Barack Obama



(b) Graph representation of RDF triples about Barack Obama

Figure 2.1 – RDF triples, extracted from DBpedia, asserting facts about Barack Obama.

Definition 1 (RDF triple) *An RDF triple is a tuple* (s, p, o) *from* $(I \cup B) \times (I) \times (I \cup B \cup L)$ *that connects a subject* s *through a predicate* p *to an object* o*.*

Definition 2 (RDF graph) *An RDF graph* $\mathcal{G}$ *is a finite set of RDF triples.*

Definition 3 (RDF dataset) *An RDF dataset is a set* $\mathcal{D} = \{\mathcal{G}_0, \langle u_1, \mathcal{G}_1\rangle, \ldots, \langle u_n, \mathcal{G}_2\rangle\}$ *where* $\mathcal{G}_0, \ldots, \mathcal{G}_n$ *are RDF graphs,* $u_1, \ldots, u_n$ *are distinct URIs, and* $n \geq 0$*. In the dataset,* $\mathcal{G}_0$ *is the default graph, and the pairs* $\langle u_i, \mathcal{G}_i\rangle$ *are named graphs, with* u_i *the name of* $\mathcal{G}_i$*.*

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

SELECT ?author ?book WHERE {
    ?author dbp:title dbr:President_of_the_United_States .
    ?author dbo:author ?book .
}
```

(a) SPARQL Query $Q1$. Returns books written by a president of the United States on DBpedia.

?author	?book
dbr:Barack_Obama	dbr:Dreams_from_My_Father
dbr:Barack_Obama	dbr:A_Promised_Land

(b) Results of executing Query $Q1$ on the RDF graph in Figure 2.1b.

Figure 2.2 – Example of a simple SPARQL query with its solutions mappings.

2.2 The SPARQL Query Language

The SPARQL [61] query language allows data consumers to query and manipulate RDF data. For example, the SPARQL query $Q1$ depicted in Figure 2.2a returns all books written by a president of the United States on DBpedia. If we execute $Q1$ on the set of RDF triples shown in Figure 2.1a, we get the results in Figure 2.2b.

The core fragment of the SPARQL query language was formalized by Perez et al. [100], and further extended by Schmidt et al. [115]. Let V be an infinite set of variables, disjoint from previous sets I, B and L. A SPARQL query is defined as a SPARQL graph pattern.

Definition 4 (Graph pattern) *SPARQL graph patterns are defined recursively as follows. (1) A triple pattern* $tp \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$ *is a SPARQL graph pattern. (2) Let* P_1 *and* P_2 *be SPARQL graph patterns. Then* $(P_1$ *UNION* $P_2)$, $(P_1$ *AND* $P_2)$ *and* $(P_1$ *OPTIONAL* $P_2)$ *are SPARQL graph patterns. (3) Let* P *be a SPARQL graph pattern and* R *a filter condition. Then* $(P$ *FILTER* $R)$ *is a SPARQL graph pattern.*

The semantics of SPARQL queries is defined in terms of mappings. A mapping is a partial function $(V) \rightarrow (I \cup B \cup L)$ that maps variables to RDF terms. The domain of a mapping μ, denoted $dom(\mu)$, is defined as the subset of V for which μ is defined. As a

naming convention, we distinguish variables from elements in $(I \cup B \cup L)$ using a leading question mark symbol. Let μ_1 and μ_2 be two mappings, μ_1 is said to be compatible with μ_2 if $\mu_1[?x] = \mu_2[?x]$ for all $?x \in dom(\mu_1) \cap dom(\mu_2)$. We write $\mu_1 \sim \mu_2$ if μ_1 is compatible with μ_2, and $\mu_1 \nsim \mu_2$ otherwise. Furthermore, $vars(tp)$ denotes all the variables in a triple pattern tp, while $\mu(tp)$ denotes the triple pattern obtained when replacing all variables $?x \in dom(\mu) \cap vars(tp)$ in tp by $\mu[?x]$.

SPARQL filter conditions are constructed using variables, RDF terms, logical connectives ($\neg$, $\wedge$ and $\vee$), inequality symbols ($<, \leq, \geq, >$), the equality symbol ($=$), unary predicates like $bound$, $isBlank$, $isIRI$, and many others [61]. In this thesis, we restrict filter conditions to the fragment of filters where the filter condition is a boolean combination of terms constructed by using the equality symbol. We write $\mu \models R$ if the mapping μ satisfies the filter condition R.

Definition 5 (Filter condition) *Filter conditions are defined recursively as follows.* (1) *Let $?x$, $?y$ be variables in V and c be an RDF term in $I \cup B \cup L$, then $?x = c$ and $?x = ?y$ are filter conditions.* (2) *Let R_1 and R_2 be filter conditions, then $(\neg R_1)$, $(R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are filter conditions.*

The semantics of SPARQL is formally defined using a compact algebra over sets of mappings. The definition of the algebraic operators join ($\bowtie$), union ($\cup$), set minus ($\setminus$), left outer join ($\leftouterjoin$) and selection (σ) is given below.

Definition 6 (SPARQL semantics) *Let Ω, Ω_1 and Ω_2 be sets of mappings, the join of, the union of, the difference between, the left outer-join of Ω_1 and Ω_2, as well as the selection over Ω are defined as:*

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \wedge \mu_1 \sim \mu_2\} \\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\} \\
\Omega_1 \setminus \Omega_2 &= \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1 \nsim \mu_2\} \\
\Omega_1 \leftouterjoin \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2) \\
\sigma_R(\Omega) &= \{\mu \in \Omega \mid \mu \models R\}
\end{aligned}$$

Let D be an RDF dataset, the evaluation of a graph pattern over D, denoted $[\![\cdot]\!]_D$, is defined below.

Definition 7 (Graph pattern semantics) *Let tp be a triple pattern, R a filter condition, P, P_1 and P_2 graph patterns. The semantics of SPARQL graph pattern evaluation over an RDF dataset D is defined as follows.*

$$\begin{aligned}
[\![tp]\!]_D &= \{\mu \mid dom(\mu) = vars(tp) \wedge \mu(tp) \in D\} \\
[\![P_1 \textit{ AND } P_2]\!]_D &= [\![P_1]\!]_D \bowtie [\![P_2]\!]_D \\
[\![P_1 \textit{ UNION } P_2]\!]_D &= [\![P_1]\!]_D \cup [\![P_2]\!]_D \\
[\![P_1 \textit{ OPTIONAL } P_2]\!]_D &= [\![P_1]\!]_D \mathbin{⟕} [\![P_2]\!]_D \\
[\![P \textit{ FILTER } R]\!]_D &= \sigma_R([\![P]\!]_D)
\end{aligned}$$

SPARQL 1.1 introduced new features such as property path and aggregate queries [61]. To better align with our contributions related to these features, we introduce SPARQL property path and aggregate queries in Section 5.2 and Section 6.2, respectively.

Part II

Providing a Fair SPARQL Query Service that Ensures Complete Results

Chapter 3

Introduction

This part focuses on our first research question: *How to provide a fair SPARQL query service for online knowledge graphs that ensures complete results?*

Public SPARQL query services depend on a fair-use policy, which is based on quotas, to prevent convoy effects [32] and ensure a fair allocation of server resources among users. To illustrate, the Wikidata SPARQL endpoint enforces a 60-second time limit on query execution time. In doing so, no query can block the server for more than 60 seconds. While fair-use policies address availability issues, they prevent SPARQL endpoints from delivering complete results. For example, Query $Q1$ in Figure 3.1, which searches Wikidata for all creative works inspired by a work of fiction, times out on the public SPARQL endpoint of Wikidata, and returns no results. Unfortunately, Query $Q1$ is not an isolated case [33]. According to Wikidata query logs [1], thousands of queries time out each month.

Delivering partial results is a severe limitation as it prevents Semantic Web applications, such as federation engines, from relying on public SPARQL endpoints. To deal with this issue, various strategies have been proposed to provide SPARQL query services that guarantee complete results [4, 23, 24, 67, 92, 134]. The key idea is to distribute the query processing load between the server and the client. The server implements only a fragment of the SPARQL algebra [61], i.e., SPARQL operators for which no query can generate convoy effects. The remaining operators are handled by the client. For instance, the TPF server exclusively manages paginated triple pattern queries [134]. As the time complexity to retrieve any page of k results is the same for all triple pattern queries, the server does not suffer from convoy effects. However, to execute any SPARQL query, the TPF client has no choice but to decompose SPARQL queries into sets of sub-queries supported by the server. As the client must send at least one HTTP request per sub-query, and transfer sub-queries results, i.e., intermediate results, such a decomposition may drastically impact query execution performance.

1. https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?creativeWork ?fictionalWork WHERE {
  ?creativeWork wdt:P144 ?fictionalWork .
  ?creativeWork wdt:P31/wdt:P279* wd:Q17537576 .
  ?fictionalWork wdt:P136 wd:Q8253
}
```

(a) $Q1$: All results

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?creativeWork ?fictionalWork WHERE {
  ?creativeWork wdt:P144 ?fictionalWork .
  ?creativeWork wdt:P31/wdt:P279* wd:Q17537576 .
  ?fictionalWork wdt:P136 wd:Q8253
} LIMIT 50
```

(b) $Q2$: First 50 results

Figure 3.1 – Creative works and the list of fiction works that inspired them on Wikidata.

For example, consider Query $Q2$ in Figure 3.1. Query $Q2$ is similar to Query $Q1$ but returns only the first 50 results of $Q1$. On the public SPARQL endpoint of Wikidata, Query $Q2$ completes in 25 seconds. To contrast, the same query requires more than 8 hours to complete using TPF [2], and 15 minutes using SaGe [3], TPF and SaGe being restricted interfaces, i.e., servers that implement only a subset of SPARQL operators to avoid convoy effects. As neither TPF nor SaGe can fully process $Q2$ on the server, both transfer many intermediate results and send many HTTP requests, drastically impacting query execution performance. While restricted interfaces ensure complete results, a fair allocation of server resources, and perform better than SPARQL endpoints under high query load [23, 24, 92], such a performance gap, without query load, makes the adoption of restricted interfaces unrealistic.

To reduce this performance gap, we need a restricted SPARQL interface that supports as many operators as possible without degrading server availability. SaGe, which is a server based on Web Preemption [92], already supports core SPARQL operators. However, SaGe does not support property path and COUNT-DISTINCT queries. Consequently, these queries are mainly executed on the client, which significantly impacts SaGe performance. Improving query execution performance for these queries is imperative as they play a crucial role in numerous use cases, such as navigating into complex hierarchies or computing valuable statistics.

The remaining of this part is organized as follows. Chapter 4 presents the state-of-the-art related to SPARQL query services and their limitations. In chapters 5 and 6, we present our extensions to the Web Preemption model to improve execution performance of COUNT-DISTINCT and property path queries, respectively. In chapter 7, we are taking our quest for performance a step further with a new algorithm to optimize join orders of SPARQL queries, especially in the presence of property paths.

2. `https://query.linkeddatafragments.org/`
3. `sage.univ-nantes.fr/`

Chapter 4

Querying Knowledge Graphs Online: State of The Art

Contents

In this chapter, we detail the different approaches that have been proposed to provide online SPARQL query services. One of the most popular interfaces for querying knowledge graphs is *SPARQL endpoints*.

4.1 SPARQL Endpoints

SPARQL endpoints are Web services that implement the SPARQL protocol [48]. The SPARQL protocol standardizes the interactions between a client and a SPARQL query service. It "describes a means for conveying SPARQL queries and updates to a SPARQL processing service and returning the results via HTTP to the entity that requested them". Many triple stores, such as Virtuoso [1], Blazegraph [2], Apache Jena [3], and Eclipse RDF4J [4], provide an HTTP interface that implements the SPARQL protocol. For example, Blazegraph is behind the public SPARQL endpoint of Wikidata [5], while Virtu-

1. https://virtuoso.openlinksw.com
2. https://blazegraph.com
3. https://jena.apache.org
4. https://rdf4j.org
5. https://www.wikidata.org/wiki/Wikidata:Main_Page

oso is behind the well-known DBpedia knowledge graph [6]. However, allowing any user to execute any SPARQL query at any time can be very costly in terms of server CPU time and memory consumption. Several studies have highlighted that public SPARQL endpoints suffer from significant performance issues under high query loads [134], leading to downtimes in the worst case [19, 132].

4.1.1 Fair-Use Policies

SPARQL endpoints execute queries using a first-come first-served execution policy. Consequently, they are exposed to the convoy effect [32], i.e., a single SPARQL query may monopolize all server resources for a long time, preventing other queries from running. To avoid the convoy effect and ensure a fair-use policy of server resources, public SPARQL endpoints enforce quotas in time and space. SPARQL queries that exceed one of these quotas are interrupted and return only partial results. For example, the DBpedia SPARQL endpoint rejects all queries running after 120 seconds and queries that return more than 10,000 results. Delivering partial results is a severe limitation as it prevents applications from relying on SPARQL endpoints. In this thesis, we propose a solution to query RDF data online that ensures complete results and a fair allocation of server resources.

4.1.2 Decomposition of SPARQL Queries

A solution for getting around quotas is to decompose SPARQL queries into smaller sub-queries that respect fair-use policies [18, 124]. Different strategies have been proposed, ranging from simple pagination to more complex query decomposition techniques. However, all of these strategies suffer from significant drawbacks. First, they need to know the server configuration in terms of quotas. Such information may be unavailable as data providers may not wish to publish the configuration of their servers. Second, they may be unable to find sub-queries that can be evaluated under quotas. Even a simple triple pattern query can take hours to complete and return millions of results. Pagination [18] is not efficiently supported by all query engines, while partition-based rewriting [124] requires extensive knowledge of the data. Finally, they may generate high network traffic, significantly degrading query processing performance. For

6. `https://www.dbpedia.org`

each sub-query, the client sends an HTTP request to the server and downloads intermediate results.

4.2 LDF Interfaces

The main challenge of decomposing SPARQL queries is ensuring that sub-queries can be completed under quotas. To tackle this issue, Linked Data Fragments (LDF) interfaces provide SPARQL query services that ensure complete results for a restricted fragment of the SPARQL algebra. Clients can then decompose SPARQL queries into sub-queries that are guaranteed to complete. The Triple Pattern Fragments [134] (TPF) approach is the first implementation of LDF. A TPF server only evaluates paginated triple pattern queries. As such queries can be evaluated in logarithmic time with respect to data size, the server does not suffer from the convoy effect. To evaluate a Basic Graph Pattern (BGP), a TPF client decomposes the BGP into a set of triple pattern queries using a similar decomposition strategy than Aranda et al. [18]. For a BGP $P = \{tp1, tp2\}$, the client first evaluates $tp1$ using the paginated server interface, retrieving pages of relevant RDF triples, and converts them into solutions mappings matching $tp1$. Then, for each solution mappings $\mu_1 \in [\![tp1]\!]_G$, it evaluates the sub-query $\mu_1(tp2)$ using the server interface, and for each $\mu_2 \in [\![\mu_1(tp2)]\!]_G$ it produces a new solution mappings $\mu = \mu_1 \cup \mu_2$ matching $tp1 \bowtie tp2$. With such a decomposition, TPF transfers many intermediate results and sends many HTTP calls, leading to poor query execution performance.

The Bindings-Restricted Triple Pattern Fragments [67] (BrTPF) approach extends the TPF server with the bind-join algorithm [59]. The bind-join algorithm allows the client to send a bag of solutions mappings alongside triple patterns, significantly reducing the number of HTTP calls. To evaluate a BGP $P = \{tp1, tp2\}$, BrTPF starts by evaluating $tp1$ as TPF. Then, for each bag $\Omega \subseteq [\![tp1]\!]_G$ of at most k solutions mappings, it evaluates the sub-query $\bigcup_{\mu_1 \in \Omega} \mu_1(tp2)$ using the server interface, and for each $\mu_2 \in \bigcup_{\mu_1 \in \Omega} [\![\mu_1(tp2)]\!]_G$ it produces a new solution mappings $\mu = \mu_1 \cup \mu_2$ matching $tp1 \bowtie tp2$. Using the bind-join algorithm, BrTPF divides by at most k the number of HTTP calls sent to evaluate a join between two triple patterns. Despite adding a new operator, the worst-case complexity of evaluating a paginated triple pattern extended with the bind-join algorithm remains logarithmic. However, even if BrTPF reduces the number of HTTP requests, joins still generate many of them, and data transfer remains the same as TPF.

To reduce the number of local joins, the Star Pattern Fragments [4] (SPF) approach proposes a new trade-off between server availability and query execution performance. Rather than decomposing SPARQL queries into sets of triple pattern sub-queries, SPF decomposes them into sets of star-shaped sub-queries. In doing so, SPF allows the client to evaluate several joins on the server, decreasing the amount of intermediate results that have to be transferred over the network and improving query execution performance. As star-shaped queries can be answered in linear time complexity [100], SPF achieves better performance than TPF and BrTPF without significantly impacting availability. However, the expressivity of the server remains limited. For queries that contain chains, which represent a significant number of queries [33, 34] including property path queries, the client has to execute local joins.

SmartKG [23] focuses on the same fragment as SPF but relies on a different strategy to improve query execution performance. Rather than evaluating star-shaped sub-queries on the server, SmartKG ships compressed and queryable graph partitions to the client that can be used to evaluate star-shaped sub-queries. Sub-queries for which there are no partitions are evaluated using a BrTPF interface. Shipping partitions is inexpensive for the server and significantly reduces the number of HTTP calls sent by the client, as local joins can be evaluated on the downloaded partitions. However, SmartKG shares the same limitations as SPF, while shipping partitions may be expensive regarding data transfer when only a few triples are relevant.

WiseKG [24] addresses the data transfer issue of SmartKG by combining SmartKG with SPF. WiseKG proposes a cost model that dynamically delegates the load between servers and clients. Based on the current server load, client capabilities, and estimates of necessary data transfer and network bandwidth, WiseKG picks the best alternative between processing star-shaped sub-queries on the client using shipped SmartKG partitions or on the server using SPF. While WiseKG significantly improves average workload completion time and reduces resource consumption, e.g., CPU usage and network traffic, WiseKG still focuses on star-shaped sub-queries only to improve query execution performance.

4.3 Web Preemption

The different LDF interfaces improve server availability by distributing the query processing load between servers and clients. Such a strategy may generate high net-

work costs as the server can only support a small fragment of the SPARQL query language without impacting availability. Thus, most of the query processing load must be handled by clients. Web Preemption [92] tackles the availability issue using a different perspective. The problem may not lie in the time it takes for a query to be executed but that it monopolizes server resources. In other words, the problem lies in the First-Come First-Served (FCFS) execution policy of SPARQL endpoints. Operating systems [16] have heavily studied FCFS scheduling policies and the convoy effect. In a system where the duration of tasks varies, a long-running task can block all others, deteriorating the average completion time for all tasks and creating a convoy effect. To avoid convoy effects, researchers designed various families of scheduling algorithms. In particular, the Round-Robin (RR) scheduling algorithm [16] provides a fair allocation of CPU between tasks, avoids the convoy effect, reduces the waiting time, and provides excellent responsiveness. RR runs a job for a given time quantum, then suspends it and switches to the next task. It repeatedly does so until all tasks are finished. Rather than a FCFS execution policy, Web Preemption proposes a new query execution model based on a Round-Robin execution policy. Web Preemption considers the SPARQL queries sent to the server as the tasks to be executed, and the Web server as the resource the tasks want to access. Unlike LDF interfaces, Web Preemption can execute any SPARQL conjunctive query with filters and unions on the server, drastically improving query execution performance. Nevertheless, a preemptable SPARQL server only supports queries that can be suspended and resumed with a logarithmic time complexity with respect to data size [92]. Queries containing property paths and aggregate functions cannot be fully processed on the server and must be decomposed by clients.

Different solutions have been proposed to improve the execution time of such queries. For example, Grall et al. demonstrate that a preemptable SPARQL endpoint supports partial aggregates [53], which can be used to reduce the amount of transferred data for SPARQL aggregate queries. However, COUNT-DISTINCT aggregate functions still generate high data transfer. In a previous work [7], we proposed an approach to improve query decomposition of property path queries with transitive closures. However, such a query decomposition still requires evaluating many joins locally on the client.

The following chapters detail our contributions to improve Web Preemption for COUNT-DISTINCT queries and property path queries with transitive closures.

Chapter 5

Online Approximative Query Processing for COUNT-DISTINCT Queries with Web Preemption

5.1 Introduction

In this chapter, we extend the Web Preemption model to improve query execution performance of SPARQL COUNT-DISTINCT queries. Grall et al. have proposed a preemptable partial aggregation operator for Web Preemption [53]. Computing partial aggregations on a preemptable server drastically reduces data transfer for most aggregate queries, while ensuring complete results. However, COUNT-DISTINCT aggregate queries still generate a large data transfer, even with a partial aggregation operator. Computing the exact number of distinct elements in a multiset requires a data transfer proportional to the number of distinct elements, which is impractical for very large datasets. For example, on Wikidata there are over a hundred million distinct entities [1].

To improve the evaluation of COUNT-DISTINCT aggregate queries, we extend the partial aggregation operator [53] with HyperLogLog++ (HLL++) sketches [74]. HyperLogLog++ is a probabilistic algorithm that can estimate the cardinality of large sets with a small amount of memory and strong guarantees on the error rate. As HLL++ supports the decomposability property of aggregate functions, it can be integrated into the partial aggregations framework [53] of Web Preemption. Compared to related Approximated Query Approaches [124], this approach ensures finding all $GroupKeys$ in a single pass, with a pre-defined and bounded error rate for each $GroupKey$. In this chapter: (1) we extend the partial aggregation operator [53]. This extension allows us to estimate the result of a COUNT-DISTINCT query with a bounded error rate and small

1. https://www.wikidata.org/wiki/Wikidata:Statistics

```
Q1:
SELECT ?c (COUNT(?o) AS ?z)
WHERE {
    ?s a ?c ; ?p ?o ; :p1 :o1 .
} GROUP BY ?c
```

(a) SPARQL query Q_1

```
Q2:
SELECT ?c (COUNT(DISTINCT(?o)) AS ?z)
WHERE {
    ?s a ?c ; ?p ?o ; :p1 :o1 .
} GROUP BY ?c
```

(b) SPARQL query Q_2

:s1 :p1 :o1 . :s1 **a** :c2, :c3 . :s2 :p1 :o1 . :s2 **a** :c1, :c3 .

(c) RDF Graph $\mathcal{G}_1$

Figure 5.1 – Aggregate queries Q_1 and Q_2 over $\mathcal{G}_1$.

memory requirements; (2) we provide additional experimental results to compare the performance of the extended operator and the previous operator [53]. Experimental results demonstrate that relying on estimates does not improve query execution time, but significantly reduces data transfer for COUNT-DISTINCT queries, and in the general case, shows that the proposed approach outperforms existing approaches used for processing aggregate queries.

The remainder of this chapter is structured as follows. Section 5.2 introduces SPARQL aggregate queries and the Web Preemption model. Section 5.3 presents the approach for processing aggregate queries using a preemptive SPARQL server. Section 5.4 introduces HyperLogLog++ and its integration in the partial aggregation operator. Section 5.5 presents the different algorithms used to implement the proposed approach. Section 5.6 presents our experimental results. Section 5.7 discusses the limitations of the current proposal. Finally, conclusions and prospects are outlined in Section 5.8.

5.2 Preliminaries

In this section, we extend the definition of the SPARQL query language presented in Section 2.2 to introduce the semantics of SPARQL aggregate queries.

5.2.1 SPARQL Aggregate Queries

The SPARQL 1.1 language [61] introduces new features for supporting aggregate queries: (1) a collection of *aggregate functions* for computing values, like `COUNT`, `SUM`, `MIN`, `MAX` and `AVG`; (2) the `GROUP BY` and `HAVING` operators. `HAVING` restricts the application of aggregate functions to groups of solutions satisfying certain conditions.

Both groups and aggregates deal with lists of expressions $\langle E_1, \ldots, E_n \rangle$, which are evaluated to v-lists, i.e., lists of values in $(I \cup B \cup L) \cup \{error\}$. More precisely, the evaluation of a list of expressions $E = \langle E_1, \ldots, E_n \rangle$, according to a mapping μ, is defined as: $[\![E]\!]^{\mu} = \langle [\![E_1]\!]^{\mu}, \ldots, [\![E_n]\!]^{\mu} \rangle$. For simplicity, lists of expressions are restricted to lists of variables. As all queries that use lists of expressions can be rewritten into queries where grouping is only allowed over lists of variables [79], this restriction does not reduce the expressive power of aggregates. Following common notations [61, 79], we formalize Group and Aggregate as follows.

Definition 8 (Group) *A group is a construct $G(E, P)$ with E a list of expressions and P a graph pattern. The evaluation $[\![G(E, P)]\!]_{\mathcal{G}}$ of a group $G(E, P)$ over an RDF graph $\mathcal{G}$ produces a set of partial functions from v-list keys (called **GroupKeys**) to multisets of mappings as follows:*

$$[\![G(E, P)]\!]_{\mathcal{G}} = \{GroupKey \mapsto \{\mu' \mid \mu' \in [\![P]\!]_{\mathcal{G}}, [\![E]\!]^{\mu'} = GroupKey\}\}$$

Definition 9 (Aggregate) *An aggregate is a construct $\gamma(F, f, P)$ with F a list of expressions, f an aggregate function and P a graph pattern. Let $\{k_1 \mapsto \omega_1, \ldots, k_n \mapsto \omega_n\}$ be the set of partial functions produced by the evaluation of $[\![G(E, P)]\!]_{\mathcal{G}}$ over an RDF graph $\mathcal{G}$ where $\langle k_1, \ldots, k_n \rangle$ are $GroupKeys$ and $\langle \omega_1, \ldots, \omega_n \rangle$ are multisets of mappings. The evaluation of $[\![\gamma(F, f, P)]\!]_{\mathcal{G}}$ maps each $GroupKey$ to a single value as follows:*

$$[\![\gamma(F, f, P)]\!]_{\mathcal{G}} = \{k_i \mapsto f(\Omega), \Omega = \{[\![F]\!]^{\mu'} \mid \mu' \in \omega_i\}\}$$

To illustrate, consider the query Q_1 of Figure 5.1a, which returns the total number of objects per class, for subjects connected to the object o_1, through the predicate p_1. P_{Q_1} = { `?s :a ?c. ?s ?p ?o. ?s :p1 :o1.`} is the graph pattern of Q_1. According to Definition 8, the evaluation of Query Q_1 over the RDF graph $\mathcal{G}_1$ given in Figure 5.1c returns:

$$\begin{aligned}[\![G(\langle \texttt{?c} \rangle, P_{Q_1})]\!]_{\mathcal{G}_1} = \{&\texttt{:c3} \mapsto \{\texttt{:c3}, \texttt{:c1}, \texttt{:c2}, \texttt{:o1}, \texttt{:c3}, \texttt{:o1}\},\\ &\texttt{:c1} \mapsto \{\texttt{:o1}, \texttt{:c3}, \texttt{:c1}\},\\ &\texttt{:c2} \mapsto \{\texttt{:o1}, \texttt{:c3}, \texttt{:c2}\}\}\end{aligned}$$

where $\langle$`?c`$\rangle$ is the list of expressions E used by the `GROUP BY` operator in Q_1. As we can see, Q_1 returns 3 different $GroupKeys$, i.e., `:c1`, `:c2` and `:c3`. For simplicity, for each $GroupKey$, we represent only the values of the $?o$ variable, as $?o$ is the only variable used by the *COUNT* function. Then, according to Definition 9, the evaluation of Q_1

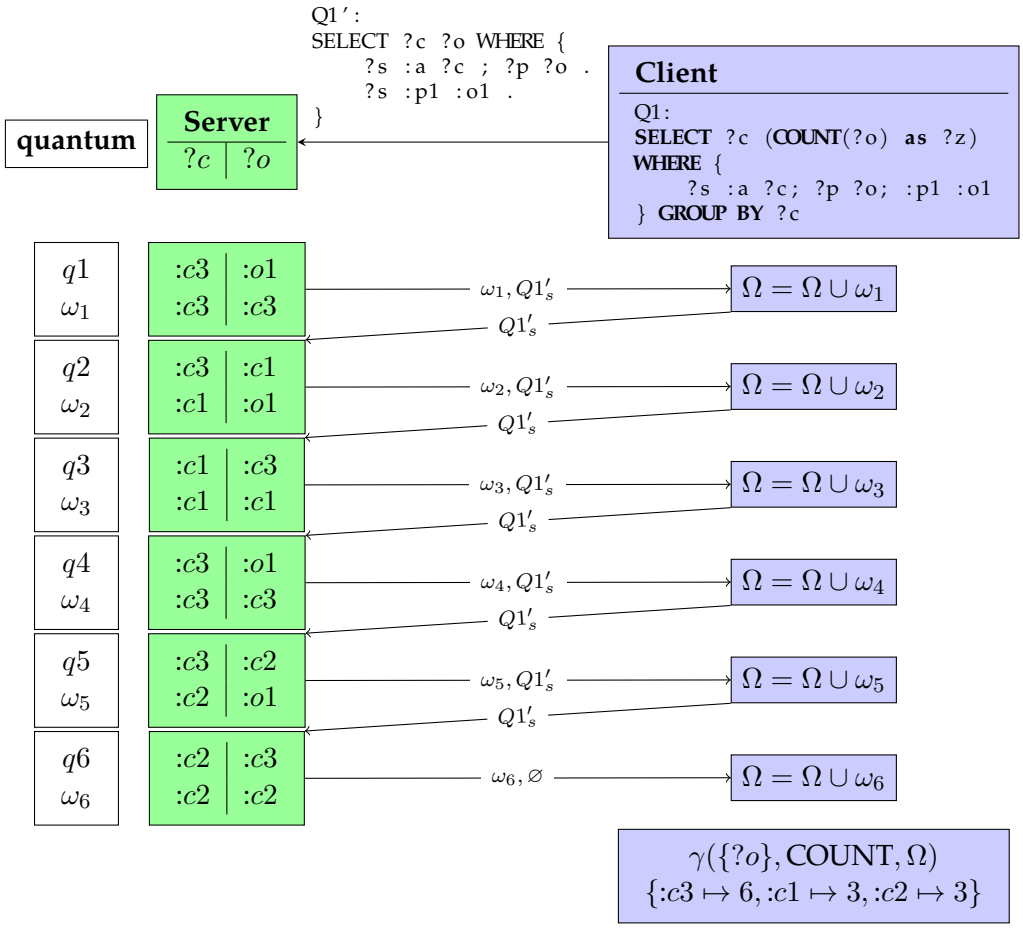


Figure 5.2 – Evaluation of Q_1 against $\mathcal{G}_1$ with regular Web Preemption [92].

over $\mathcal{G}_1$ returns:

$$[\![\gamma(\langle \texttt{?o} \rangle, \texttt{COUNT}, P_{Q_1})]\!]_{\mathcal{G}_1} = \{\texttt{:c3} \mapsto 6, \texttt{:c1} \mapsto 3, \texttt{:c2} \mapsto 3\}$$

where `COUNT` is the aggregate function f and $\langle \texttt{?o} \rangle$ is the list of expressions F used by f.

5.2.2 Web Preemption and SPARQL Aggregate Queries

Web Preemption [92] is the capacity of a Web server to suspend a running SPARQL query after a fixed quantum of time and resume the next waiting query. When suspending a query Q, a preemptable server saves the internal state of all operators of Q in a saved plan Q_s that is sent to the Web client. The Web client can continue the exe-

cution of Q by sending Q_s back to the server. When reading Q_s, the server restarts the query Q from where it has been stopped. As a preemptable server can restart queries from where they have been stopped and makes progress at each quantum, it eventually delivers complete results.

However, Web Preemption comes with overheads. The time taken by the suspend and resume operations represents the overhead in time of a preemptable server. The size of Q_s represents the overhead in space of a preemptable server and may be transferred over the network each time the server suspends a query. To be tractable, a preemptable server has to minimize these overheads.

For this purpose, a preemptable server only implements SPARQL operators that can be saved and resumed in constant time, i.e., preemptable operators. Based on the definition of tuple-at-a-time and full-relation operators [51], SPARQL operators can be classified into two groups: mapping-at-a-time and full-mappings operators.

Mapping-at-a-time operators such as `SCAN`, `JOIN`, `UNION` or `BIND` are implemented on the server. As they just need to manage one mapping at a time [51], these operators can be saved and resumed in constant time[2] [92]. On the other hand, full-mappings operators require "seeing all or most of the mappings in memory at once" [51]. Consequently, they cannot be saved and resumed in constant time and are implemented on the Web client. For example, in the worst case, `ORDER BY` is a full-mappings operator, i.e., the server has no choice but to materialize all the mappings before sorting them. This case typically arises when the `ORDER BY` operator is not combined with a `LIMIT k` operator, or the server does not have the required sorted indexes.

To evaluate a query that contains full-mappings operators, the Web client must decompose it into a set of sub-queries supported by the server, evaluate each sub-query separately and recombine the intermediate results to produce the final query result. Such a decomposition can be extremely costly in terms of data transfer, number of HTTP calls to the server, and execution time. Unfortunately, aggregate queries require a server-side operator that belongs to the full-mappings operators [51]. Consequently, there is no support on the server and aggregate queries must be decomposed.

Figure 5.2 illustrates how Web Preemption processes Query Q_1 of Figure 5.1a over the RDF graph $\mathcal{G}_1$. First, the Web client sends the BGP of Q_1 to the server, i.e., the query Q'_1 = `SELECT ?c ?o WHERE { ?s :a ?c; ?p ?o; :p1 :o1 }`. Let us suppose that the

2. A `SCAN` can be resumed in $O(log(|D|))$ with B-Tree indexes on SPO, POS and OSP, where $|D|$ is the size of the dataset D.

query Q'_1 requires six quanta to complete. At the end of each quantum q_i, the Web client receives the mappings ω_i and asks for the next results ($next$ link). Then, when all mappings are obtained, the Web client computes $\gamma(\langle \texttt{?o} \rangle, \texttt{COUNT}, \bigcup_i \omega_i)$. As a result, to compute the three mappings $\{\texttt{:c3} \mapsto 6, \texttt{:c1} \mapsto 3, \texttt{:c2} \mapsto 3\}$, the server transferred $6 + 3 + 3 = 12$ mappings to the Web client.

In a more general way, to evaluate $[\![\gamma(F, f, \Omega)]\!]_{\mathcal{G}}$, the Web client first asks a preemptable Web server to evaluate $[\![P]\!]_{\mathcal{G}} = \Omega$. Then the server transfers incrementally Ω, and finally, the Web client evaluates $\gamma(F, f, \Omega)$ locally. The main problem with this evaluation is that the size of Ω is usually much bigger than the size of $\gamma(F, f, \Omega)$.

Reducing data transfer requires reducing $|[\![P]\!]_{\mathcal{G}}|$, which is impossible without deteriorating the completeness of the answer. Therefore, the only way to reduce data transfer when processing aggregate queries is to process the aggregates on the preemptable server. However, in the worst case, the operator we need to evaluate SPARQL aggregates is a *full-mappings* operator, as it requires to materialize $|[\![P]\!]_{\mathcal{G}}|$, hence it cannot be suspended and resumed in constant time.

Problem Statement: Define a preemptable aggregation operator γ such that the complexity in time and space of suspending and resuming γ is bounded in constant time. [3]

5.3 Computing Partial Aggregations with Web Preemption

To build a preemptable operator for SPARQL aggregates, our approach relies on two key ideas: (1) Web Preemption naturally creates a partitioning of mappings over time. Thanks to the decomposability of aggregate functions [146], partial aggregations can be computed server-side on each partition of mappings and recombined on the Web client; (2) the quantum size can be adjusted for aggregate queries to control the size of partial aggregations.

In the following, the decomposability property of aggregate functions is presented, as well as how this property is used in the context of Web Preemption.

3. For simplicity, only aggregate queries with Basic Graph Patterns and no OPTIONAL clauses are considered.

Table 5.1 – Decomposition of SPARQL aggregate functions w/o the DISTINCT modifier.

(a) Aggregate functions without the DISTINCT modifier

SPARQL Aggregate functions					
	COUNT	SUM	MIN	MAX	AVG
f_1	COUNT	SUM	MIN	MAX	SaC
$v \diamond v'$	$v + v'$		$\min(v, v')$	$\max(v, v')$	$v \oplus v'$
h	Id				$(x, y) \mapsto x/y$

(b) Aggregate functions with the DISTINCT modifier

SPARQL Aggregate functions				
	$\texttt{COUNT}_D$	$\texttt{SUM}_D$	$\texttt{AVG}_D$	$\texttt{COUNT}^{\epsilon}_D$
f_1	CT			HLL^{ϵ}_{add}
$v \diamond v'$	$v \cup v'$			HLL^{ϵ}_{merge}
h	COUNT	SUM	AVG	HLL^{ϵ}_{count}

5.3.1 Decomposability of Aggregate Functions

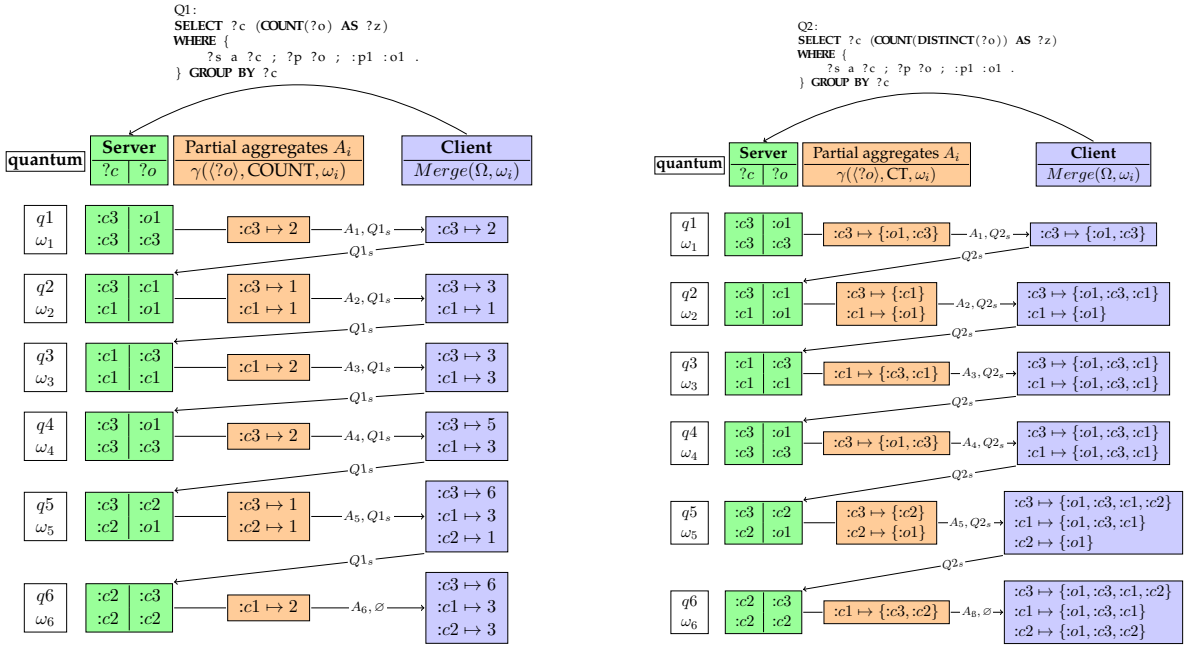
Traditionally, the decomposability property of aggregate functions [146] ensures the correctness of the distributed computation of the aggregates [77]. The decomposability property is adapted for SPARQL aggregate queries in Definition 10.

Definition 10 (Decomposability) *An aggregate function f that uses a list of expressions F is decomposable if for all non-empty multisets of solutions mappings Ω_1 and Ω_2, there exists a (merge) operator $\diamond$, a post-process function h and an aggregate function f_1 such that:*

$$\gamma(F, f, \Omega_1 \uplus \Omega_2) = \{GroupKey \mapsto h(v_1 \diamond v_2) \mid GroupKey \mapsto v_1 \in \gamma(F, f_1, \Omega_1), \\ GroupKey \mapsto v_2 \in \gamma(F, f_1, \Omega_2)\}$$

In Definition 10, $\uplus$ denotes the union operator on multisets [79]. Abusing the notation, we replace the graph pattern P in Definition 9 by a multiset of solutions mappings Ω knowing that $[\![P]\!]_{\mathcal{G}} = \Omega$. Table 5.1 gives the decomposition of all SPARQL aggregate functions, where Id denotes the identity function and $\oplus$ is the *point-wise sum of pairs*, i.e., $(x_1, y_1) \oplus (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$.

To illustrate, consider the function $f = \texttt{COUNT}$ with $F = \langle \texttt{?c} \rangle$ and an aggregate query $\gamma(F, f, \Omega_1 \uplus \Omega_2)$ such that $\gamma(F, f, \Omega_1) = \{\texttt{:c1} \mapsto 2\}$ and $\gamma(F, f, \Omega_2) = \{\texttt{:c1} \mapsto 5\}$. The intermediate aggregation results for the COUNT function can be merged using the arithmetic addition operator, i.e., $\{:c1 \mapsto 2 \diamond 5 = 2 + 5 = 7\}$.



(a) Evaluation of Q_1

(b) Evaluation of Q_2

Figure 5.3 – Evaluation of Q_1 and Q_2 over $\mathcal{G}_1$ with a partial aggregation operator.

Decomposing `SUM`, `COUNT`, `MIN` and `MAX` is simple as partial aggregation results only need to be merged to produce the final query results. However, decomposing `AVG` and aggregate functions that use the `DISTINCT` modifier are more complex. Two auxiliary aggregate functions have been introduced, called `SaC` (SUM-and-COUNT) and `CT` (Collect), respectively. The `SaC` function collects the information required to compute an average, while the `CT` function collects a set of distinct values. They are defined as follows: $\texttt{SaC}(X) = \langle \texttt{SUM}(X), \texttt{COUNT}(X)\rangle$ and $\texttt{CT}(X)$ is the base set of X as defined in section 5.2. For instance, the aggregate function of the query $Q = \gamma(\langle \texttt{?o}\rangle, \texttt{COUNT}_D, \Omega_1 \uplus \Omega_2)$ is decomposed as $Q' = \texttt{COUNT}(\gamma(\langle \texttt{?o}\rangle, \texttt{CT}, \Omega_1) \cup \gamma(\langle \texttt{?o}\rangle, \texttt{CT}, \Omega_2))$.

5.3.2 Partial Aggregations with Web Preemption

Using a preemptive Web server, the evaluation of a graph pattern P over an RDF graph $\mathcal{G}$ naturally creates a partitioning of mappings $\omega_1, \ldots, \omega_n$ over time, where ω_i is produced during the quantum q_i. Intuitively, a partial aggregation A_i, formalized in Definition 11, is obtained by applying an aggregate function on ω_i.

Definition 11 (Partial aggregation) *Let F be a list of expressions, f an aggregate function and $\omega_i \subseteq [\![P]\!]_{\mathcal{G}}$ such that $[\![P]\!]_{\mathcal{G}} = \bigcup_{i=1}^{i=n} \omega_i$ where n is the number of quanta required to evaluate P over the RDF graph $\mathcal{G}$. A partial aggregation A_i is defined as $A_i = \gamma(F, f, \omega_i)$.*

Because partial aggregations are computed over a single partition of mappings, any partial aggregation can be implemented on the server using a mapping-at-a-time operator. Such an operator does not require materializing intermediate results as partial aggregations are sent to the Web client at the end of each quantum. Consequently, SPARQL aggregate queries that use a partial aggregation operator can be suspended and resumed in constant time. Finally, to return complete and correct results, the Web client simply merges partial aggregations, i.e., $[\![\gamma(F, f, P)]\!]_{\mathcal{G}} = h(A_1 \diamond A_2 \diamond \cdots \diamond A_n)$.

To illustrate, Figure 5.3a depicts the evaluation of Query Q_1 over the RDF graph $\mathcal{G}_1$ using Web Preemption with partial aggregations. Let us suppose that Q_1 needs six quanta $q_1, \ldots, q_6$ to complete, each of which produces two mappings. On the one hand, for each quantum q_i, the server collects the mappings in w_i. Then, it computes the partial aggregation $A_i = \gamma(\langle \texttt{?o} \rangle, \texttt{COUNT}, \omega_i)$ over w_i, and sends A_i to the Web client. On the other hand, the Web client collects and merges all A_i using the $\diamond$ operator. Once all partial aggregations have been merged, the Web client produces the final results by applying the post-process function h.

Figure 5.3b depicts the execution of Query Q_2 under the same conditions as Query Q_1. As we can see, combining the `DISTINCT` modifier with the `COUNT` aggregation function requires transferring more data to the Web client. However, a reduction in data transfer is still observable compared with transferring all ω_i for quanta q_1, q_2, q_3, q_4, q_5 and q_6. The server does not transfer duplicates by quantum and $GroupKey$.

The duration of a quantum seriously impacts query processing using partial aggregations. If the evaluation of Query Q_1 over the RDF graph $\mathcal{G}_1$ required twelve quanta rather than six in Figure 5.3a, each producing a single mapping, then partial aggregations would have been useless. If the server only required two quanta producing six mappings each, then only two partial aggregations $A_1 = \{\texttt{:c3} \mapsto 3, \texttt{:c1} \mapsto 3\}$ and $A_2 = \{\texttt{:c3} \mapsto 3, \texttt{:c2} \mapsto 3\}$ would have been sent to the Web client, significantly reducing data transfer. If the quantum were infinite, the whole aggregation would have been produced on the server, and data transfer would have been optimal. Overall, for a SPARQL aggregate query, the larger the quantum, the smaller the data transfer and execution time.

5.4 COUNT-DISTINCT SPARQL Aggregate Queries

COUNT-DISTINCT aggregate queries count the number of distinct elements in the multiset obtained after grouping. Query Q_2 of Figure 5.1b is an example of a COUNT-DISTINCT aggregate query.

As illustrated in Figure 5.3b, processing COUNT-DISTINCT aggregate queries requires transferring all elements from the server to the Web client before counting them. Moreover, these elements can be transferred several times if the query is processed over several quanta. For example, `:o1` and `:c3` for the $GroupKey$ `:c3` in Figure 5.3b are transferred twice. Consequently, for each $GroupKey$, computing an exact count requires an amount of memory, and thus data transfer, proportional to the cardinality of the underlying multiset. Such a data transfer is prohibitive and does not scale to large datasets.

To address this issue, we propose to estimate the number of distinct elements in a multiset rather than computing the exact count. Several probabilistic algorithms have been proposed to estimate large cardinalities with a bounded memory [47, 50, 141]. For example, the LinearCounting algorithm [141] achieves good accuracy, regardless of the cardinality. However, this algorithm is not attractive for very large cardinalities, as it requires too much memory for an accurate estimate [74]. Compared to the LinearCounting algorithm, the HyperLogLog (HLL) algorithm [50] is efficient for very large cardinalities, both in terms of space complexity and accuracy. For instance, HLL can estimate cardinalities greater than 10^9 with a typical error rate of 2%, using only 1.5KBytes of memory. However, HLL fails to estimate the number of distinct elements in small multisets. Moreover, the HLL algorithm is not memory efficient. No matter if the cardinality to be estimated is small, it uses the maximum amount of memory specified by the user, e.g., 1.5KBytes for an error rate of 2%.

In the context of SPARQL aggregate queries, a good estimator must be accurate on both small and large cardinalities, and adapt its memory usage to cardinality. Indeed, aggregate queries deal with $GroupKeys$ that may have millions of distinct values as well as just a few. To fit these criteria, we use HyperLogLog++ (HLL++) [74], an adaptive counting algorithm that combines the HLL and the LinearCounting algorithms. Because the LinearCounting algorithm is more efficient for small cardinalities than HLL, HLL++ relies on the LinearCounting algorithm to estimate small cardinalities, and automatically switches to the HLL algorithm for larger cardinalities. Finally, HLL++ supports the decomposability property of aggregate functions. Consequently, HLL++ can

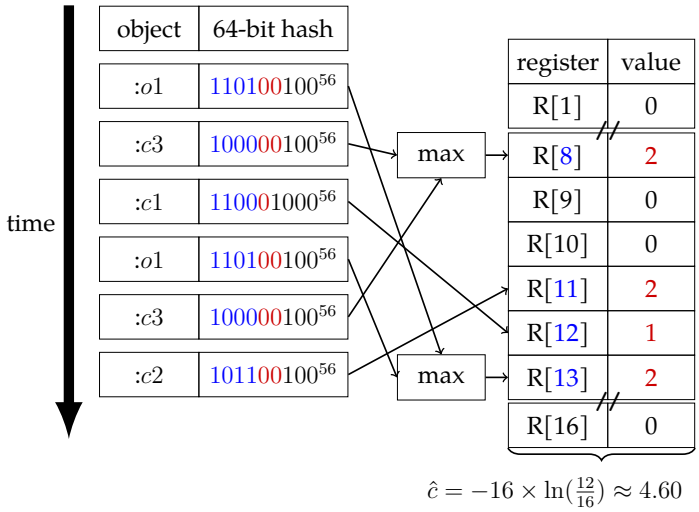


Figure 5.4 – Approximate COUNT-DISTINCT for the $GroupKey$:c3 of Query Q_2, on the RDF graph $\mathcal{G}_1$, using *HLL++* with an error rate of 26%.

be used to extend the partial aggregation operator proposed for Web Preemption [53]. A Web client merging partial aggregations based on HLL++ can now compute the number of distinct elements with a bounded error rate and bounded data transfer for each $GroupKey$.

5.4.1 HyperLogLog++

HyperLogLog++ (HLL++) is a probabilistic data structure that behaves like a set with two main operations:

1. HLL^{ϵ}_{add} for adding a new element e to the set.
2. HLL^{ϵ}_{count} for estimating the cardinality of the set with a fixed error rate ϵ.

The payload of a HLL++ set is an array R of m registers, denoted $R[1], \ldots, R[m]$. The number of registers m is defined using the formula $m = (1.04/\epsilon)^2$ where ϵ is the error rate we wish to guarantee [74, 50]. To add an element e into a HLL++ set H^{ϵ}, where ϵ is the selected error rate, e is first mapped to a 64-bit hash value $h(e)$. The first $p = log_2(m)$ bits of $h(e)$ represents the index i of R to update. The number of leading zeros k located just after the first p bits are stored in $R[i]$, if $k > R[i]$.

To estimate the cardinality of H^{ϵ}, HLL++ relies both on the HyperLogLog (HLL) and the LinearCounting algorithms. It first uses HLL to estimate the cardinality of H^{ϵ}. If the estimate is greater than a pre-defined threshold [50], HLL++ uses the estimate

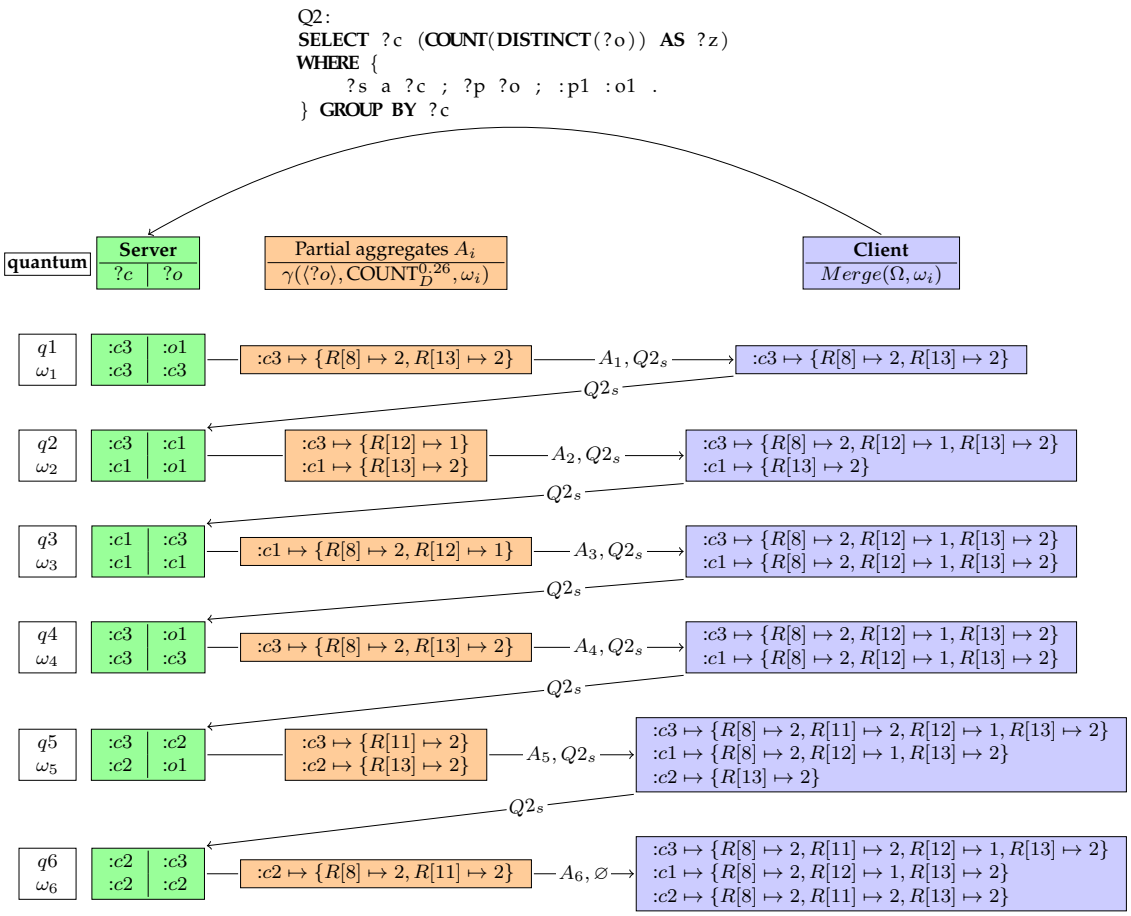


Figure 5.5 – Evaluation of Query Q_2 on the RDF graph $\mathcal{G}_1$ using *HLL++* with an error rate of 26%.

of the HLL algorithm, otherwise it uses the estimate of the LinearCounting algorithm.

To estimate the cardinality of H^ϵ, the HLL algorithm relies on the idea that, in a uniformly distributed multiset of 64-bit hash values, i.e., numbers, long runs of leading zeros, in the binary representation of these numbers, are less likely and indicate a larger cardinality. Based on this observation, if the maximum number of leading zeros is k, a good estimate of the number of distinct elements is 2^k. Because a single measurement would have too much variability, HLL divides the elements into m registers and then computes the cardinality estimate as the harmonic mean of the m registers. This technique known as stochastic averaging operates as a variance reduction device [50], which increases the quality of estimates.

On its side, the LinearCounting algorithm relies on the fraction of empty registers to estimate the cardinality of H^ϵ. Using the LinearCounting algorithm, an estimate of

the cardinality of H^ϵ is given by the equation $-m \times \ln(V)$ where V is the number of empty registers divided by the total number of registers m [141].

To illustrate how HLL++ works, consider the example of Figure 5.3b where the $GroupKey$:c3 is incrementally filled with elements :o1, :c3, :c1, :o1, :c3 and :c2 to finally obtain four distinct elements. In Figure 5.4, the same elements are added to an HLL++ set $H_{c3}^{0.26}$ where the error rate $\epsilon = 26\%$ and the number of registers $m = (1.04/0.26)^2 = 16$. Each element is mapped to a 64-bit hash value. The first $p = log_2(16) = 4$ bits are represented in blue and used for identifying the register $R[i]$ to update. The number of leading zeros k after the first $p = 4$ bits are highlighted in red and used to update $R[i]$, if $k > R[i]$. Once all the elements are inserted in $H_{c3}^{0.26}$, HLL++ estimates the number of distinct elements for the $GroupKey$:c3 using either the HLL or the LinearCounting algorithm. Because, in our example, the number of distinct elements is small, HLL++ uses the LinearCounting algorithm [74], and estimates the number of distinct elements as $-16 \times \ln(12/16) \approx 4.60$.

5.4.2 Partial Aggregations and HyperLogLog++

To estimate the number of distinct elements in a multiset, with a fixed error rate ϵ, we introduce a new aggregate function $COUNT_D^\epsilon$ (cf Table 5.1). To follow the partial aggregations model, $COUNT_D^\epsilon$ has to provide an aggregate function f_1, a merge operator $\diamond$ and a post-process function h as defined in Definition 10. Functions f_1, $\diamond$ and h of $COUNT_D^\epsilon$ are respectively mapped to HLL_{add}^ϵ, HLL_{merge}^ϵ and HLL_{count}^ϵ, where HLL_{merge}^ϵ merges two HLL++ sets H_1^ϵ and H_2^ϵ of m registers into a new HLL++ set H_3^ϵ such that $H_3^\epsilon.R[i] = \max(H_1^\epsilon.R[i], H_2^\epsilon.R[i])$ for $i \in 1..m$.

Figure 5.5a illustrates how a Web client computes Query Q_2 over the RDF graph $\mathcal{G}_1$ with a fixed error rate $\epsilon = 26\%$ using $COUNT_D^{0.26}$. At each quantum q_i, two new mappings are produced in ω_i. For each $GroupKey$ in ω_i, the server creates a HLL++ set. During the first quantum, two mappings with two different objects :o1 and :c3 are produced for the $GroupKey$:c3. Using the $HLL_{add}^{0.26}$ operation, :o1 and :c3 are assigned to registers 13 and 8, respectively. Both $R[13]$ and $R[8]$ are updated from 0 to 2 as both :o1 and :c3 hash values have two leading zeros. At the end of the quantum, registers are sent to the Web client.

Compared to the HyperLogLog algorithm, HLL++ does not necessarily send all the registers to the Web client. To fit the *memory efficiency* criteria, HLL++ can store the array R using either a sparse or a full representation [74]. The sparse representation

is used when most of the registers are empty and avoid transferring all registers to the Web client. Thus, for an error rate of 2%, the 1.5KBytes per $GroupKey$ is just a worst-case space complexity for HLL++. Typically, for small sets, the data transfer will be at most equivalent to transferring the sets.

To go back to our example, when the Web client receives the registers, it uses the $HLL^{0.26}_{merge}$ operation to merge the incoming registers with the local ones. The Web client repeats the same process for all quanta until the query complete. When the query completes, the Web client uses the $HLL^{0.26}_{count}$ operation to estimate the number of distinct elements per $GroupKey$.

The duration of a quantum has a significant impact on the data transfer. Long quanta reduce the amount of transferred data as HLL++ sets are better used. However, long quanta are also likely to gather many $GroupKeys$ that require each to store a HLL++ set. Even if HLL++ is efficient in terms of space complexity and can adapt its memory usage, gathering many $GroupKeys$ may exhaust the memory of the server. This issue is already pointed out as the many-distinct count problem [128]. In the context of Web Preemption, this issue can be avoided by limiting the memory dedicated to the aggregation results, so that a quantum only deals with a bounded number of $GroupKeys$. Once the limit is reached, even if the quantum is not exhausted, the query is suspended and the partial aggregations are sent to the Web client. Of course, such an approach just moves the many-distinct count problem to the Web client. However, the Web client memory is not a shared resource.

5.5 Implementing Decomposable Aggregate Functions

Algorithm 1 presents the general algorithm to compute partial aggregations on a preemptable server. To evaluate an aggregate query Q, the algorithm starts by building the physical query plan of Q, i.e., a pipeline of preemptable iterators (Lines 2-3), that will be consumed by Algorithm 1. Algorithm 2 defines a new preemptable iterator for computing aggregate functions. When the GetNext() method is called, the new iterator consumes a solution mappings μ from its predecessor (Line 3), and computes aggregate functions on μ (Line 5). As aggregate functions are computed one mapping at a time, this iterator is preemptable, i.e., it can be saved and resumed in constant time. During a quantum, Algorithm 1 consumes solutions mappings from the pipeline of iterators, and merges them into Ω (Lines 9-17), using the $Merge$ operation defined in Algorithm 3.

Algorithm 1: Server-side evaluation of partial aggregations

Require: $quantum$: The duration of a quantum, $pageSize$: The maximum size of a result page.

Input: Q: A SPARQL aggregate query.

```
1  Function EvalQuery(Q):
2      if Q is a suspended query then iterator ← Resume(Q)
3      else iterator ← ParseQuery(Q)
4      E ← GROUP BY expressions of Q
5      A ← Aggregate functions of Q
6      Ω ← ∅
7      done ← False
8      try:
9          EvalQuantum( quantum ):
10             repeat
11                 μ ← iterator.GetNext()
12                 non interruptible
13                     if μ ≠ nil then
14                         Merge(E, A, Ω, {μ})
15                         μ ← nil
16                     else done ← True
17             until done ∨ Size(Ω) ≥ pageSize
18     catch QuantumExhausted:
19         if μ ≠ nil then Merge(E, A, Ω, {μ})
20     finally:
21         if done then Q_s ← nil
22         else Q_s ← Suspend(iterator)
23         return (Ω, Q_s)
```

The $Merge(E, A, X, Y)$ operation merges two sets of solutions mappings X and Y. For each $\mu \in X$, it finds a $\mu' \in Y$ that has the same $GroupKey$ as μ (Line 3). Then, Algorithm 3 iterates over all aggregation results in μ' (Lines 4-10) to merge them with their equivalent in μ, using the different merge operators defined in Table 5.1.

When the quantum is exhausted, the server waits for the critical section of Algorithm 1 to complete. Thus, no solution mappings is discarded, and the merge operation is guaranteed to be applied to every solution mappings. The critical section can block the program for at most the time to merge a single solution mappings in Ω, which can be done in constant time. Finally, Algorithm 1 suspends the query Q, and sends the suspended query Q_s alongside the partial aggregates Ω to the Web client (Lines 21-23).

Algorithm 2: Server-side preemptable partial aggregation iterator

Require: I: a preemptable iterator, E: *GROUP BY* expressions, A: a set of 3-tuple (F, f, v) where f is an aggregate function, F a list of expressions and v a variable to bind the result of f.

Data: μ_c: the last element read from I_p.

```
1  Function GetNext():
2      if ∀(F, f, v) v ∈ dom(μ_c) then return μ_c
3      μ_c ← I.GetNext()
4      if μ_c ≠ nil then
5          ComputeAggregates(E, A, μ_c)
6          return μ_c
7      else return nil
8  Procedure ComputeAggregates(E, A, μ):
9      foreach (F, f, v) ∈ A do
10         Ω ← γ(F, f, {μ}) ; μ[v] ← Ω[[E]^μ]
11 Function Save():
12     return μ_c
13 Procedure Load(μ):
14     μ_c ← μ
15     if μ_c ≠ nil then
           ComputeAggregates(E, A, μ_c)
```

To avoid the many-count distinct problem, i.e., exhaust server memory, the size of Ω is bounded to a predefined constant $PageSize$. If the size of Ω exceeds $PageSize$ (Line 17), Algorithm 1 stops computing new aggregation results and the query is suspended.

The evaluation of SPARQL aggregates on the server requires defining different parameters: the duration of a quantum, the maximum space allocated to the aggregation results, and the error rate ϵ when the $COUNT^{\epsilon}_{D}$ function is used. Defining these parameters is left to the server administrator.

As the server only computes partial aggregations, it relies on the Web client to compute SPARQL aggregates, as shown in Algorithm 4. To execute a SPARQL aggregate query Q, the Web client first decomposes Q into Q' to replace the `AVG` aggregate function and the `DISTINCT` modifier as described in Section 5.3.2. Then, the Web client submits Q' to the SaGe server S, and follows the `next` links sent by S to fetch and merge all query results, following the Web Preemption model (Lines 6-9). Finally, the Web client transforms the set of partial aggregation results sent by the server to produce the final aggregation results (Line 10). For each solution mappings $\mu \in \Omega$, the Web client applies the appropriate h function for each of the aggregate functions, as defined in Table 5.1.

Algorithm 3: Merge two sets of solutions mappings, Y into X

Input: X, Y: two sets of solutions mappings, E: *GROUP BY* expressions, A: a set of 3-tuple (F, f, v) where f is an aggregate function, F a list of expressions and v a variable to bind the result of f.

Procedure $Merge(E, A, X, Y)$**:**
  foreach $\mu \in X$ **do**
    foreach $\mu' \in Y$ ***with*** $[\![E]\!]^{\mu} = [\![E]\!]^{\mu'}$ **do**
      foreach $(F, f, v) \in A$ **do**
        if $f \in \{\texttt{COUNT}, \texttt{SUM}\}$ **then** $\mu[v] \leftarrow \mu[v] + \mu'[v]$
        if $f = \texttt{SaC}$ **then** $\mu[v] \leftarrow \mu[v] \oplus \mu'[v]$
        if $f = \texttt{MIN}$ **then** $\mu[v] \leftarrow Min(\mu[v], \mu'[v])$
        if $f = \texttt{MAX}$ **then** $\mu[v] \leftarrow Max(\mu[v], \mu'[v])$
        if $f = \texttt{CT}$ **then** $\mu[v] \leftarrow \mu[v] \cup \mu'[v]$
        if $f = \texttt{COUNT}^{\epsilon}_{D}$ **then** $\mu[v] \leftarrow HLL^{\epsilon}_{merge}(\mu[v], \mu'[v])$
  foreach $\mu' \in Y, \nexists \mu \in X, [\![E]\!]^{\mu} = [\![E]\!]^{\mu'}$ **do**
    $X \leftarrow X \cup \{\mu'\}$

5.6 Experimental Study

The experimental study aims to answer the following questions: (1) What is the data transfer reduction obtained with partial aggregations? (2) What is the speed-up obtained with partial aggregations? (3) What is the impact of the duration of a quantum on data transfer and execution time? (4) Does estimating the result of COUNT-DISTINCT queries reduce data transfer? (5) Does the observed error rate match the theoretical guarantees provided by the HLL++ algorithm?

5.6.1 Experimental Setup

The partial aggregations approach has been implemented as an extension of the SaGe query engine [4], an open source implementation of Web Preemption. The SaGe server has been extended with the new operator described in Algorithm 1. Python SaGe-agg and SaGe-approx Web clients have been extended with Algorithm 4. SaGe-agg uses the $COUNT_D$ function to compute COUNT-DISTINCT queries, while SaGe-approx uses the $COUNT^{\epsilon}_{D}$ function. The source code of the experimental study as well as all configuration files are available in the project repository at `https://github.com/JulienDavat/sage-agg-experiments`.

4. `https://sage.univ-nantes.fr`

Algorithm 4: Client-side evaluation of partial aggregations

Input: Q: A SPARQL aggregate query, S: The URL of a SaGe server.

```
1  Function EvalQuery(Q, S):
2      E ← GROUP BY expressions of Q
3      A ← Aggregate functions of Q
4      Q' ← DecomposeQuery(Q)
5      Ω ← ∅
6      repeat
7          (Ω', Q') ← Evaluate Q' at S
8          Merge(E, A, Ω, Ω')
9      until Q' = nil
10     ComputeAggregates(Ω, A)
11     return Ω
12 Procedure ComputeAggregates(Ω, A):
13     foreach μ ∈ Ω do
14         foreach (F, f, v) ∈ A do
15             if f = AVG then
16                 (s, c) ← μ[v] ; μ[v] ← s/c
17             if f = SUM_D then
18                 μ[v] ← SUM(μ[v])
19             if f = AVG_D then
20                 μ[v] ← AVG(μ[v])
21             if f = COUNT_D then
22                 μ[v] ← |μ[v]|
23             if f = COUNT^ε_D then
24                 μ[v] ← HLL^ε_count(μ[v])
```

Table 5.2 – Statistics of RDF datasets used in the experimental study.

	# Triples	# Subjects	# Predicates	# Objects	# Classes
BSBM-10	4,987	614	40	1,920	11
BSBM-100	40,177	4,174	40	11,012	22
BSBM-1k	371,911	36,433	40	86,202	103
DBpedia 3.5.1	100,000,000	2,835,701	35,168	26,840,695	342

Dataset and Queries: The *SP* workload in our experiments is composed of 18 SPARQL aggregate queries extracted from SPORTAL queries [70]. Most of the extracted queries use the DISTINCT modifier, and time out on the DBpedia public SPARQL endpoint [70]. As depicted in Figure 5.6, the SPORTAL queries return $GroupKeys$ with different numbers of distinct values; from one to several million on DBpedia. Such a property is important to demonstrate that HLL++ is accurate on both small and large multisets. To study the impact of the DISTINCT modifier on query execution, we define a new workload, denoted *SP-ND*, by removing all DISTINCT modifiers from the *SP* workload.

Both the *SP* and the *SP-ND* workloads are run on synthetic and real-world datasets. For the synthetic datasets, we use the Berlin SPARQL Benchmark (BSBM) to generate three datasets of increasing size: BSBM-10, BSBM-100 and BSBM-1k. For the real-world dataset, we use a fragment of DBpedia. The statistics of each dataset are detailed in Table 5.2.

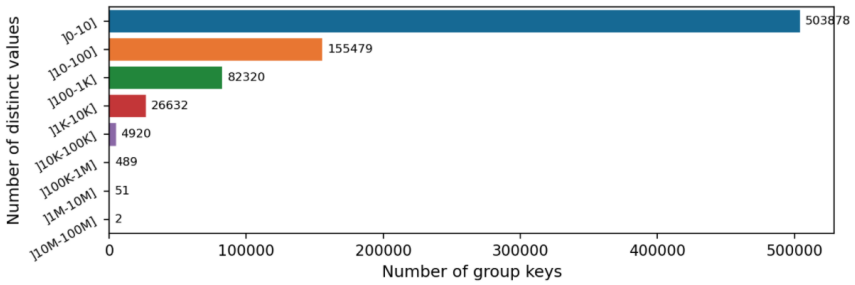


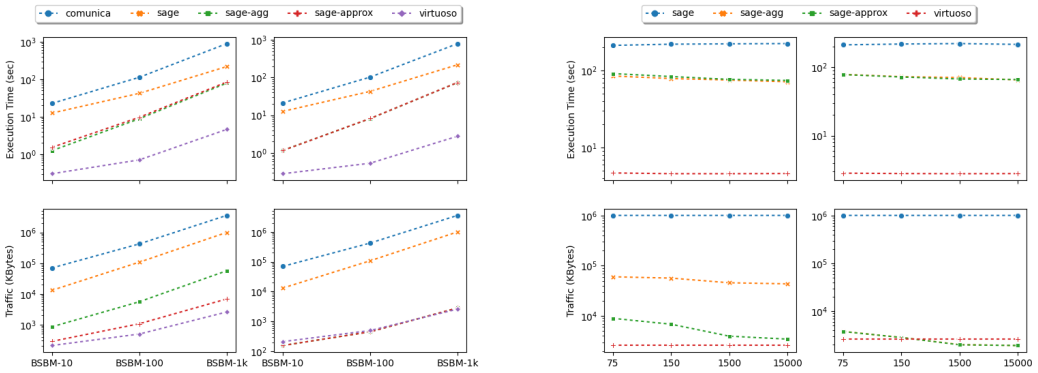
Figure 5.6 – Number of $GroupKeys$ for the SPORTAL queries on DBpedia according to the number of distinct values.

Approaches: The following approaches are compared:

- *SaGe* is a SPARQL query engine that implements the Web Preemption model [92]. The SaGe server is configured with a $PageSize$ of 10MBytes. The data are stored in a SQLite database, with B-tree indexes on (SPO), (POS) and (OSP).
- *SaGe-agg* is an extension of SaGe with partial aggregations [53]. To be fairly compared with SaGe, SaGe-agg is configured as SaGe.
- *SaGe-approx* is an extension of SaGe-agg with HLL++ sketches. To be fairly compared with SaGe and SaGe-agg, SaGe-approx is configured as SaGe. To compute COUNT-DISTINCT queries, SaGe-approx uses an error rate $\epsilon = 2\%$.
- *TPF* is the TPF query engine [134]. The TPF server is configured with a page size of 10000 mappings and without Web caches. Data are stored using the HDT format. The TPF client is Comunica [126] (v1.9.4).
- *Virtuoso* is the Virtuoso SPARQL endpoint [46] (v7.2.4). Virtuoso is configured **without quotas** and with a *single thread* so that it delivers complete results and can be fairly compared with other engines.

Servers configurations: All experiments have been run on the Google Cloud Platform, on a n2-highmem-4 machine with 4 vCPU, 32 GBytes of RAM and a SSD local disk of 375 GBytes.

Evaluation Metrics: Presented results correspond to the average of three successive executions of the query workloads. (i) *Data transfer*: is the number of bytes transferred to the Web client when evaluating a query. (ii) *Execution time*: is the time between the start of the query and the production of the final results by the Web client. (iii) *Error rate*:



(a) Data transfer and execution time on BSBM-10, BSBM-100 and BSBM-1K

(b) Impact of the quantum on BSBM-1K

Figure 5.7 – BSBM experiments on the *SP* (left) and *SP-ND* (right) workloads.

is defined as the difference between the real cardinality c and the estimated cardinality $\hat{c}$: $(1 - (\min(c, \hat{c})/\max(c, \hat{c})) \times 100$.

5.6.2 Experimental Results

Data Transfer and Execution Time over BSBM Datasets

Figure 5.7a presents the data transfer and execution time over BSBM-10, BSBM-100 and BSBM-1k. In this experiment, the SaGe server is configured with a time quantum of 150ms. The plots on the left detail the results for the SP workload, while the plots on the right detail the results for the SP-ND workload.

As expected, Virtuoso without quota performs best regarding data transfer and execution time. On the other hand, TPF offers the worst performance as it does not support projections nor joins on the server-side. As a result, TPF transfers many intermediate results and sends many HTTP requests to the server, which significantly impacts query execution time. Although both SaGe and TPF evaluate SPARQL aggregate queries on the client-side, SaGe delivers better performance than TPF because it supports projections and joins on the server.

Compared to SaGe, SaGe-agg and SaGe-approx drastically reduce data transfer but do not improve the execution time because partial aggregations do not increase the scanning speed on the disk. When comparing the performance of SaGe-agg and SaGe-approx on the two workloads, we can observe that query processing without the DIS-

TINCT modifier (on the right) is much more efficient in terms of data transfer than with the DISTINCT modifier (on the left).

Without the DISTINCT modifier, SaGe-agg and SaGe-approx are equivalent and transfer only one number per $GroupKey$, per quantum. Consequently, they can achieve performance that are close to Virtuoso. Note that if the data transfer for Virtuoso is a bit larger than SaGe-agg and SaGe-approx, it is only because of the output format used by the different endpoints. In the best case, SaGe-agg and SaGe-approx can only be as good as Virtuoso.

For queries that use the DISTINCT modifier, SaGe-agg has to transfer all terms observed during a quantum. The only optimization that can be done to reduce data transfer is to remove the duplicates observed during the same quantum. However, those observed during different quanta cannot be removed. Compared to SaGe-agg, SaGe-approx significantly improves the evaluation of COUNT-DISTINCT queries regarding data transfer. For each $GroupKey$, the HLL++ algorithm transfers at most m registers (integers). For an error rate of 2%, HLL++ uses $m = 4096$ registers, which represents a worst-case data transfer of 1.5KBytes [74]. For $GroupKeys$ that return a very large number of different terms, 1.5KBytes is not much compared to what it would cost to send all the terms to the Web client. For $GroupKeys$ that return a small number of different terms, HLL++ transfers only the used registers. For instance, if a $GroupKey$ returns 10 different terms, HLL++ will only transfer at most 10 registers.

Data Transfer and Execution Time over DBpedia

To confirm the results observed on the synthetic datasets, we ran the SP workload on a fragment of DBpedia, using both SaGe-agg, SaGe-approx and Virtuoso. The quantum for SaGe-agg and SaGe-approx has been set to 30 seconds. The results are shown in Figure 5.8, where the queries (Q2, Q3, Q4, Q5, Q7, Q8, Q9, Q10, Q12, Q13, Q15, Q16) labeled in blue are the ones that use the DISTINCT modifier.

As expected, Virtuoso delivers the best performance in terms of data transfer and execution time. In terms of execution time, the differences between Virtuoso and both SaGe-agg and SaGe-approx are mainly due to a lack of query optimization in the SaGe-agg and SaGe-approx implementations; no projection push-down, no merge-joins, etc. In terms of data transfer, Virtuoso is optimal as it computes the full aggregations on the server side and transfers only the final results. Compared to Virtuoso, SaGe-agg and SaGe-approx perform only partial aggregations on the server side. Nevertheless,

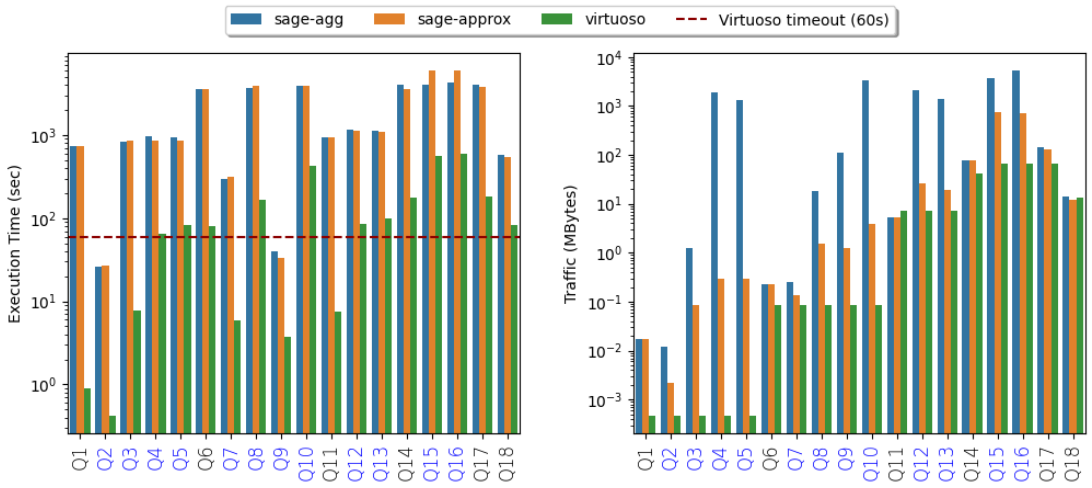


Figure 5.8 – Performance, in terms of execution time and data transfer, obtained when running the SP workload on the DBpedia dataset.

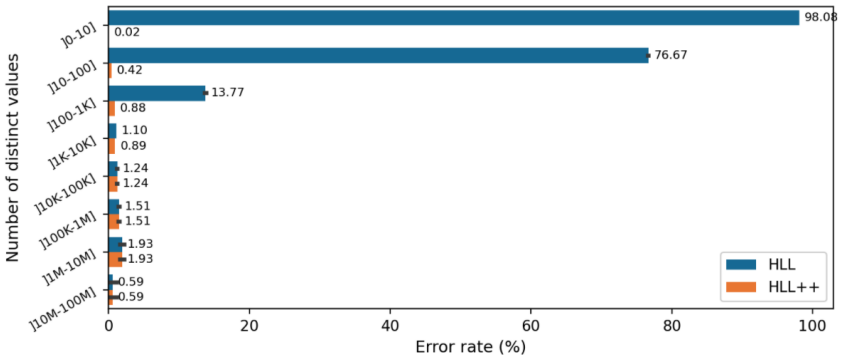


Figure 5.9 – Average error rate for the $GroupKeys$ of the SP workload queries on DBpedia according to the number of distinct values.

Virtuoso cannot ensure that all queries terminate under quotas. The red dotted line in Figure 5.8 corresponds to a quota of 60s. As we can see, queries Q5, Q6, Q8, Q10, Q12, Q13, Q14, Q15, Q16, Q17 and Q18 do not terminate, i.e., two thirds of the queries are interrupted after 60s and return **no results**.

Compared to Virtuoso, the SaGe server does not interrupt queries. The queries are just suspended after a time quantum and resumed later. Consequently, both SaGe-agg and SaGe-approx ensure the termination of all queries. Finally, as expected, SaGe-approx drastically improves performance in terms of data transfer on large RDF datasets.

Error Rates over DBpedia

SaGe-approx approximates the result of COUNT-DISTINCT queries; hence, there is a potential for error. To ensure that the theoretical guarantees on the error rate hold in practice, we measured the error rate for each $GroupKey$ returned by the queries of the SP workload on DBpedia. To compute the error rate, we used SaGe-agg as the ground truth. In Figure 5.9, $GroupKeys$ are grouped according to the number of distinct values returned and the average error rate is computed for each group. As expected, although HLL is a powerful approximate algorithm on large cardinalities, it fails on small cardinalities. Compared to HLL, HLL++ is a good estimator for the result of SPARQL COUNT-DISTINCT queries. By adapting the algorithm to compute the estimate according to the cardinalities, HLL++ achieves an error rate lower than 2% for both small and large cardinalities.

Impact of Time Quantum

To study the impact of the quantum on data transfer and query execution time, the two workloads have been run with different time quanta. Figure 5.7b reports the results of running SaGe, SaGe-agg, SaGe-approx and Virtuoso with a quantum of 75ms, 150ms, 1.5sec and 15sec on BSBM-1k. The plots on the left detail the results for the SP workload, and on the right, the SP-ND workload.

As we can see, increasing the quantum does not significantly improve the execution time. The speed of scans does not change whatever the value of the quantum. However, increasing the quantum reduces the data transfer for SaGe-agg and SaGe-approx on both workloads. Indeed, increasing the quantum allows a better use of partial aggregations. The less often a query is interrupted, the less likely it is to transfer the same $GroupKeys$ multiple times. That is why there is a significant drop between 150ms and 1.5sec in Figure 5.7b. With a quantum of 1.5sec, queries are interrupted 10 times less often than with a quantum of 150ms. Between 75ms and 150ms, queries are only interrupted half as often; consequently, the improvement is not as important as between 150ms and 1.5sec. Finally, with a quantum of 15sec, data transfer is optimal as all queries terminate between 1.5 and 15 seconds.

Finally, we can observe that SaGe-agg is less impacted by the quantum duration than SaGe-approx. Even if higher quanta allow to deduplicate more terms, the number of elements transferred by SaGe-agg remains important and dominates the data transfer.

5.7 Discussion

Experimental results demonstrate that using probabilistic data structures to compute SPARQL COUNT-DISTINCT queries significantly reduces data transfer. However, the current implementation still has poor performance in terms of execution time, which limits its application to very large knowledge graphs such as Wikidata or DBpedia. As mentioned in the experimental study, these performance issues are due to a lack of query optimization on the SaGe server. The simple application of state-of-art optimization techniques, including filter and projection push-down, aggregate push-down, or merge-joins should significantly improve performance.

Moreover, the current approach only proposes to improve the evaluation of COUNT-DISTINCT queries. To evaluate AVG-DISTINCT and SUM-DISTINCT queries, the server must still transfer all the elements to the Web client. Unfortunately, to the best of our knowledge, no probabilistic data structure currently supports estimating a distinct sum.

Finally, to avoid the many-count distinct problem, we currently rely on Web Preemption. By limiting the memory dedicated to the aggregation results, we ensure that a quantum only processes a limited number of $GroupKeys$. Such a solution has several drawbacks. First, it prevents us from using large quantum. Indeed, queries that return a large number of $GroupKeys$ will reach the memory limit before the quantum ends. As a result, the HLL++ sets will be less well utilized, and queries will require more quanta to complete, which means more HTTP calls, more data transfer, and worse execution time. Secondly, it just shifts the problem on the Web client. To address the many-count distinct problem, different approaches [143, 128] propose to make many HLL sketches share the same registers. By sharing registers, the server could deal with more $GroupKeys$ before exhausting its memory, but none of these approaches propose solutions to handle HLL++ sketches. However, as HLL++ sketches rely both on HLL and the LinearCounting algorithm, it should be possible to adapt the HLL++ algorithm so that HLL registers are shared between different HLL++ sketches.

5.8 Conclusion

In this chapter, we have extended the partial aggregation operator [53] in order to improve the evaluation of COUNT-DISTINCT aggregate queries. We have demonstrated that the decomposability property of the HyperLogLog++ algorithm can be

used to integrate HyperLogLog++ sketches in our framework. We have proved experimentally that using HyperLogLog++ sketches drastically reduce data transfer for SPARQL COUNT-DISTINCT queries. Compared to related approaches, the presented solution ensures that all $GroupKeys$ are discovered in a single pass with strong guarantees on the error rate.

The next step is to extend this approach to handle large knowledge graphs. One way to scale up is to parallelize the evaluation of SPARQL aggregate queries. Currently, Web Preemption does not support intra-query parallelization techniques. Defining how to suspend and resume parallel scans is clearly part of our research agenda. Finally, addressing the many-count distinct problem on the server could reduce the data transfer and the memory consumption on both the server and the Web client.

Chapter 6

Processing SPARQL Property Path Queries Online with Web Preemption

Contents

6.1 Introduction

In this chapter, we extend the Web Preemption model to improve query execution performance of SPARQL property path queries.

Property paths were introduced in SPARQL 1.1 [61] to add extensive navigational capabilities to the SPARQL query language. They allow to write sophisticated navigational queries over Knowledge Graphs (KGs). SPARQL queries with property paths are widely used. For instance, they represent 38% of the entire log of Wikidata [34]. However, executing these complex queries against online public SPARQL services is challenging, mainly due to quotas enforcement that prevent queries from delivering

complete results [70, 92, 101]. In this work, we focus on *how to execute SPARQL property path queries online and get complete results?*

The problem of executing property path queries online and getting complete results has already been studied in the context of Triple Pattern Fragment (TPF) interfaces [68, 134] and Web Preemption [7]. Current approaches decompose property path queries into many triple pattern or BGP queries that are guaranteed to terminate. However, such an approach generates many sub-queries, which significantly degrades query execution performance.

To tackle this issue, we extend a preemptable SPARQL server with a preemptable Partial Transitive Closure (PTC) operator based on a depth-limited search algorithm. We show that a Web client using the PTC operator can compute SPARQL 1.1 property path queries online and get complete results without decomposing queries. In this chapter: (1) we show how to build a PTC preemptable operator; (2) we show how a Web client can use Partial Transitive Closures to compute transitive closures; (3) we compare the performance of our PTC approach with existing restricted interfaces and SPARQL 1.1 servers. Experimental results demonstrate that our approach outperforms existing restricted interfaces regarding the number of HTTP calls, data transfer, and query execution time.

This chapter is organized as follows. Section 6.2 gives a quick reminder of Web Preemption and presents property paths. Section 6.3 presents the PTC approach and algorithms. Section 6.4 presents our experimental results. Finally, the conclusion is outlined in Section 6.5.

6.2 Web Preemption and Property Paths

This section recalls the principle of Web Preemption and provides a brief definition of SPARQL property path queries, as well as our problem statement.

Web Preemption [92] is the capacity of a Web server to suspend a running SPARQL query after a fixed quantum of time and resume the next waiting query. When suspending a query Q, a preemptable server saves the internal state of all operators of Q in a saved plan Q_s and sends Q_s to the Web client. The Web client can then continue the execution of Q by sending Q_s back to the server. When reading Q_s, the server restarts the query Q from where it has been stopped. As a preemptable server can restart queries

from where they have been stopped and makes progress at each quantum, it eventually delivers complete results.

However, Web Preemption comes with overheads. The time taken by the suspend and resume operations represents the overhead in time of a preemptable server. The size of Q_s represents the overhead in space of a preemptable server and may be transferred over the network each time the server suspends a query. To be tractable, a preemptable server has to minimize these overheads.

A simple triple pattern query can be suspended in constant time [92], i.e., we just need to store the last triple scanned in Q_s. Assuming that a dataset D is indexed with traditional B-Tree indexes on SPO, POS and OSP, the query can be resumed in $O(\log(|D|)$ given the last triple scanned, where $|D|$ is the size of the dataset D. Many operators such as join, union, projection, bind, and most filters can be saved and resumed in constant time as they just need to manage *one-mapping-at-a-time*. The preemptable SPARQL server processes these operators.

However, some operators need to materialize intermediate results and cannot be saved in constant time. For example, the "ORDER BY" operator needs to materialize the results before sorting them. Such operators are classified as *full-mappings* and are processed by the Web client. For example, to process an "ORDER BY" all results are first transferred to the Web client that finally sorts them. If delegating some operators to the Web client allows effective processing of any SPARQL queries, it has a cost in terms of data transfer, number of HTTP calls, and query execution time.

Unfortunately, to compute property path expressions with transitive closures, we need a server-side operator that belongs to the class of *full-mappings* operators. While transitive closures can be computed using either graph-traversal algorithms or recursive queries, both have a space complexity that is at least linear with respect to the size of the graph.

SPARQL Property Path Query (PPQ): A property path query is a SPARQL query with at least one property path pattern (PPP). A PPP is a tuple in $(I \cup B \cup V) \times E \times (I \cup B \cup L \cup V)$ where E is the set of property path expressions defined by the following grammar [81]: $e := a \mid e^- \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^+ \mid e^* \mid e? \mid !a_1, \ldots, a_k \mid !a_1^-, \ldots, a_k^-$ where $a, a_1, \ldots, a_k \in I$. This work assumes non-transitive property path expressions to be evaluated using traditional SPARQL algebra operators [61]. Thus, our proposal focuses only on transitive property path expressions, e.g., e^*. However, nested transitive

closures, e.g., $(ab^*)^*$, are not considered and will be the subject of future work.

Following the Web Preemption model, a BGP containing a property path expression with a transitive closure is fully processed by the Web client following the decomposition approach. A solution to drastically reduce the number of HTTP calls is to extend a preemptable server with a transitive closure operator such that BGP containing property path patterns can be processed on the server. However, algorithms that implement the transitive closure, such as DFS and BFS, are not preemptable, i.e., cannot be suspended and resumed in constant time.

Problem Statement: Is it possible to define a preemptable operator capable of adding support for transitive closures on the server?

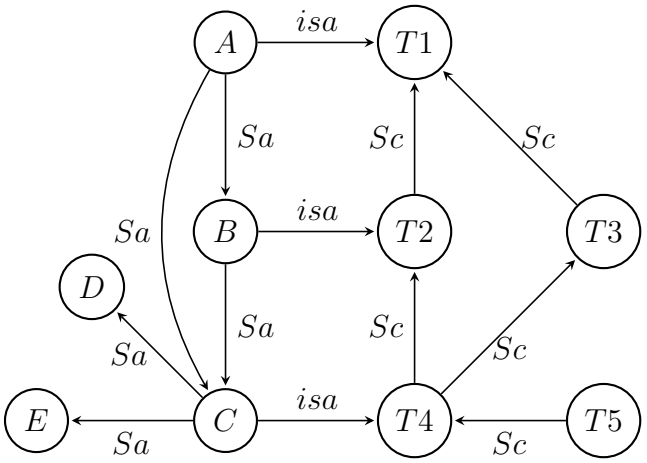
6.3 The Partial Transitive Closure Approach

To compute SPARQL 1.1 property paths online and deliver complete results, our approach relies on two key ideas:

1. Thanks to the ability of Web Preemption to save and load iterators, it is possible to implement a preemptable Partial Transitive Closure (PTC) operator. A PTC operator computes the transitive closure of a relation but cuts the exploration of the graph at a depth k. Nodes that are visited at depth k are called frontier nodes. However, such an operator cannot compute property path expressions as defined in SPARQL 1.1, i.e., transitive closures may be incomplete and return duplicates.
2. By sending frontier nodes to the Web client, it is possible to restart the evaluation of a property path query from the frontier nodes. Consequently, a Web client using the PTC operator can fully compute SPARQL 1.1 property paths. Such a strategy to compute property paths outperforms existing fair query services as queries are evaluated on the server without any joins on the Web client.

6.3.1 The PTC Operator

The *Partial Transitive Closure* $PTC(v, p, k)$ for a starting node v, a non-transitive property path expression p, and a depth k, returns all pairs (u, d) where u is a node, such that it exists a path from v to u that conforms to the expression $(p)^d$ with $d \leq k$ and d is minimal. The *frontier nodes* for a $PTC(v, p, k)$ are the nodes reached at depth k, i.e., $\{u \mid (u, d) \in PTC(v, p, k) \wedge d = k\}$.



(a) Graph D

```
SELECT ?x ?y ?z WHERE {
  ?x   :isa   :T1  . #tp1
  ?x   :sa+   ?y   . #tp2
  ?y   :isa   ?y1  . #tp3
  ?y1  :sc+   ?z   . #tp4
}
```

(b) Query $Q3$

Figure 6.1 – Graph D and Query $Q3$.

Algorithm 5: ALP auxiliary function

```
1  Let eval(v, p) be the function that returns all terms reachable from the RDF term v,
   by going through a path that matches the non-transitive path expression p.
2  Let MaxDepth be the depth limit
3  Function ALP(v:term, p:path):
4      R ← ∅ // set of terms
5      V ← ∅ // set of pairs (Term, Integer)
6      ALP(v, p, R, V)
7      return R
8  Function ALP(v, p, R, V):
9      S ← [(v, 0)] // stack of terms
10     while S ≠ ∅ do
11         (u, d) ← S.pop()
12         R.add(u)
13         V.add((u, d))
14         if d ≥ MaxDepth then continue
15         X ← eval(u, p)
16         forall x ∈ X do
17             if ∄(x, d') ∈ V, d' ≤ d then
18                 S.add((x, d + 1))
```

To illustrate, consider the $PTC(A, Sa, 2)$ that returns all nodes reachable from A through a path that conforms to the expression $(Sa)^d$ with $d \leq 2$. On the graph D of Figure 6.1a, $PTC(A, Sa, 2)$ = { $(B, 1)$, $(C, 1)$, $(D, 2)$, $(E, 2)$ } where both D and E are frontier nodes. If $PTC(n, p, k)$ returns no frontier nodes then the transitive closure is complete, i.e., k was large enough to capture the transitive closure for parameters v and p. Otherwise, frontier nodes are used by the Web client to continue the evaluation of the transitive closure until no new frontier nodes are discovered. In our context, the depth limit k ($maxDepth$) is fixed by the preemptable SPARQL endpoint administrator and can be seen as a global variable of the preemptable server.

To implement a PTC operator, we rely on a depth-limited search algorithm [108] [1]. A depth-limited search (DLS) is fundamentally a depth-first search where the search space is limited to a maximum depth. Algorithm 5 redefines the ALP auxiliary function of the SPARQL 1.1 specification [61] to follow our definition of PTC.

To avoid counting beyond a Yottabyte [20], each node is annotated with the depth at which it has been reached (Line 13). A node is revisited only if it is reached with a shortest path (Line 17). Compared to an existential semantics [20] where nodes can be visited only once, the time complexity is degraded because nodes can be revisited at most k times. However, using an existential semantics does not allow to ensure the PTC semantics [130]. To illustrate, consider $PTC(A, Sa, 2)$ evaluated previously. Under an existential semantics, starting at node A, node B is first visited at depth 1, then C at depth 2 and both B and C are marked as visited. As C cannot be revisited at depth 1, $PTC(A, Sa, 2)$ returns pairs $\{(B, 1), (C, 2)\}$. In spite of there is a path from A to D and E that match $(Sa)^2$, nodes D and E are not returned. Moreover, C appears as a frontier node and will be explored by the Web client whereas it is not a frontier node.

6.3.2 pPTC: A Preemptable PTC Iterator

The most crucial element of an iterative DLS is the stack of nodes to explore. To build a preemptable iterator based on the DLS, its stack must be saved and resumed in constant time. To achieve this goal, we do not push nodes on the stack but iterators that are used to expand nodes and explore the graph.

Algorithm 6 presents our preemptable PTC iterator, called $pPTC$. To illustrate how a property path query is evaluated using $pPTC$, suppose that the server is processing Query $Q3$ with the physical query plan of Figure 6.2. When the third index loop join iterator is first activated, it pulls the bag of mappings $\mu = \{ ?x \mapsto A, ?y \mapsto C, ?y1 \mapsto T4 \}$ from its left child. Then, it applies μ to $tp4$ in order to generate the bounded pattern $b =$ `T4 Sc+ ?z`, creates a $pPTC$ iterator to evaluate b and calls the $GetNext()$ operation of the $pPTC$ iterator, i.e., its right child.

To expand a node v, $pPTC$ creates an iterator $iter = createIter(v, p, ?o)$. Each time $iter.GetNext()$ is called, it returns a solution mappings μ_c where $\mu_c[?o]$ is the next node reachable from v through a path that conforms to p. In Figure 6.2 the first time the $GetNext()$ operation of the $pPTC$ iterator is called, it expands the node $T4$. Expanding

1. Iterative Deepening Depth-First Search (IDDFS) can also be used, but IDDFS re-traverse same nodes many times.

Algorithm 6: A *preemptable PTC iterator*, evaluating a kleene star expression without nested stars

Require:
p: path expression without stars
v: RDF term
μ: set of mappings
tp_{id}: path pattern identifier

Data:
$MaxDepth$: depth limit
S: empty stack of preemptable iterators
V: set of pairs (RDF term, Integer)
CT: empty set of control tuples

```
1  Function Open():
2      iter ← createIter(v, p, ?o)
3      S.push(iter.save())
4      μ_c ← nil
5      iter ← nil
6  Function Save():
7      if iter ≠ nil then
8          S.push(iter.save())
9      return S, path, tp_id, μ_c
10 Function Load(S', path', tp_id', μ_c'):
11     S ← S'
12     path ← path'
13     tp_id ← tp_id'
14     μ_c ← μ_c'
15 Function GetNext():
16     while S ≠ ∅ do
17         if μ_c = nil then
18             while μ_c = nil and S ≠ ∅ do
19                 iter ← S.pop().load()
20                 μ_c ← iter.getNext()
21             if μ_c = nil then return nil
22         non interruptible
23             if iter ≠ nil then
24                 S.push(iter.save())
25                 iter ← nil
26             n ← μ_c[?o]
27             if ∃(n, d) ∈ V, d ≤ |S| then
28                 continue
29             V.add((n, |S|))
30             if |S| < MaxDepth then
31                 child ← createIter(n, p, ?o)
32                 S.push(child.save())
33             CT.add((tp_id, μ, (n, |S|)))
34             solution ← μ ∪ μ_c ; μ_c ← nil
35             return solution
36     return nil
```

$T4$ with $p = Sc$ is equivalent to evaluating the triple pattern `T4 Sc ?z`. Consequently, $pPTC$ calls the function $createIter(T4, Sc, ?z)$ to create a $ScanIterator$ on the top of the stack S and calls its $GetNext()$ operation to retrieve the first child of $T4$, i.e., $T2$. When $pPTC$ wants to expand $T2$, the iterator used to expand $T4$ is saved, and a new iterator is created at the top of the stack. As depicted in Figure 6.2, compared to a traditional DLS like Algorithm 5 (Lines 16-18) the siblings of $T2$ are not stored on the stack before expanding $T2$. Because a preemptable iterator is used to explore $T4$, it can be resumed later to continue the exploration of $T2$ siblings, i.e., only iterators need to be saved, one for each node on the current path. As the current path is bounded by k, i.e. $|S| \leq k$, and a preemptable iterator can be saved and resumed in $O(|Q| \log(|D|))$ where $|Q|$ is the

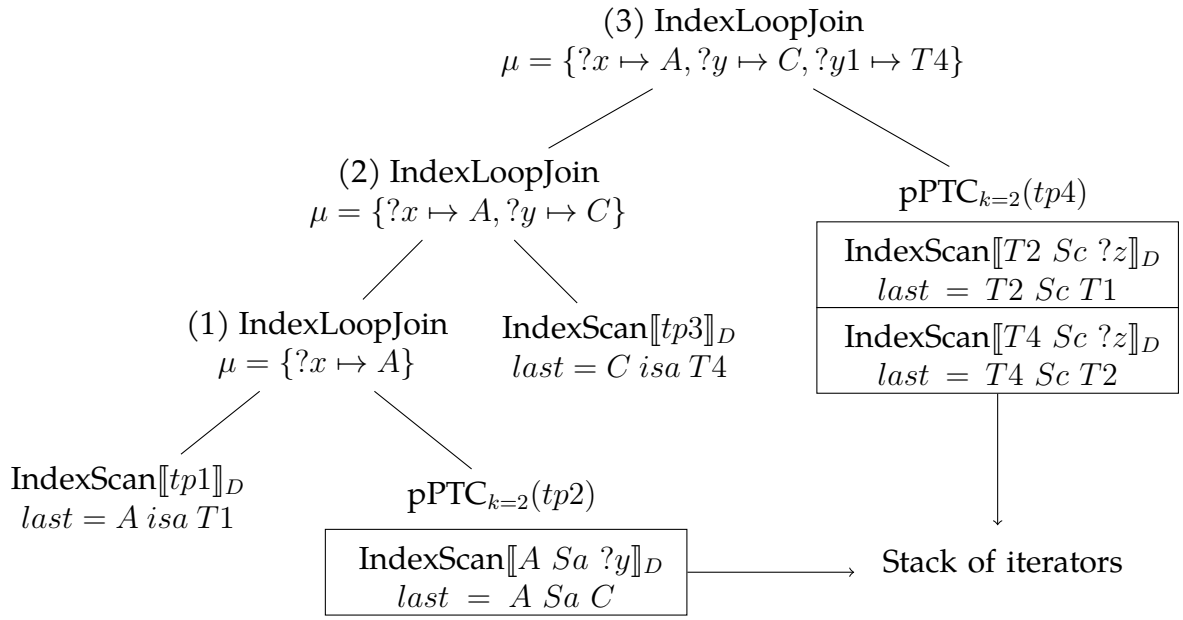


Figure 6.2 – Physical execution plan of $Q3$ with iterators internal state for $k = 2$.

size of the query and $|D|$ the size of the graph [92], a $pPTC$ iterator can be suspended and resumed in $O(k \times |p| \log(|D|))$ where $|p|$ is the number of predicates in the path expression p.

Quanta and complexities During one quantum, the $pPTC$ iterator maintains a structure to keep track of visited nodes with their corresponding depth. To be preemptable, this structure has to be flushed at the end of each quantum, keeping only the stack of iterators between two quanta. In the worst case, we can consider visited nodes as always empty. In this case, the $pPTC$ iterator enumerates all simple paths leading to a $\#P$ complexity [20]. In the best case, the $pPTC$ iterator has the same time complexity as PTC.

Controls for the PTC-Client To allow a Web client to resume a property path query and continue the evaluation beyond frontier nodes, visited nodes are contextualized and sent to the Web client. For each visited node, a $pPTC$ iterator generates a *control tuple* $ct = (tp_{id}, \mu, (n, d))$ (Line 33) where tp_{id} is the identifier of the property path pattern that produced ct, μ is the current mappings when ct has been produced and (n, d) is a pair representing a visited node with its depth. For example, suppose that the server is processing Query $Q3$ using the physical query plan of Figure 6.2. When the first index

Algorithm 7: PTC-Client

Data:
$MaxDepth$: depth limit
$FIFO$: empty queue of tuples (query Q, ptc_{id}, frontier node n)
V: maps each ptc_{id} to a set of pairs (node, depth)
R: empty multi-set of solution mappings

```
1  Function EvalClient(query):
2      FIFO.enqueue((query, nil, nil))
3      while FIFO ≠ ∅ do
4          (Q, ptc_id, n) ← FIFO.dequeue()
5          if ∃(n', d') ∈ V[ptc_id], n' = n ∧ d' < MaxDepth then continue
6          (ω, ct) ← ServerEval(Q)
7          R ← R ∪ ω
8          for (tp_id, μ, vc) ∈ ct do
9              ptc'_id ← hash(μ, tp_id)
10             for (node, depth) ∈ vc do
11                 if ∃(n', d') ∈ V[ptc'_id], n' = node then
12                     V[ptc'_id].add(node, min(d', depth))
13                 else
14                     V[ptc'_id].add(node, depth)
15                     if depth = MaxDepth then
16                         Q_e ← ExpandQuery(Q, tp_id, μ, node)
17                         FIFO.enqueue(Q_e, ptc'_id, node)
18     return R
```

loop join is activated, it pulls the bag of mappings $\{ ?x \mapsto A \}$ from its left child, and next calls the $GetNext()$ operation of the $pPTC$ iterator, i.e., its right child. In this context, the $pPTC$ iterator explores node A at depth 1 by evaluating `A Sa ?y`, returns the bag of mappings $\{ ?x \mapsto A, ?y \mapsto C \}$ and generates the control tuple $(TP2_{id}, \{?x \mapsto A\}, (A, 1))$.

All control tuples generated during a quantum are stored in a shared memory dedicated to a query during a quantum. At the end of the quantum, control tuples and solutions mappings are sent to the Web client. In order to reduce data transfer associated with control tuples, control tuples $ct_1, \ldots, ct_n$ that share the same tp_{id} and μ are grouped together into a tuple $(tp_{id}, \mu, [(n_1, d_1), \ldots, (n_n, d_n)])$.

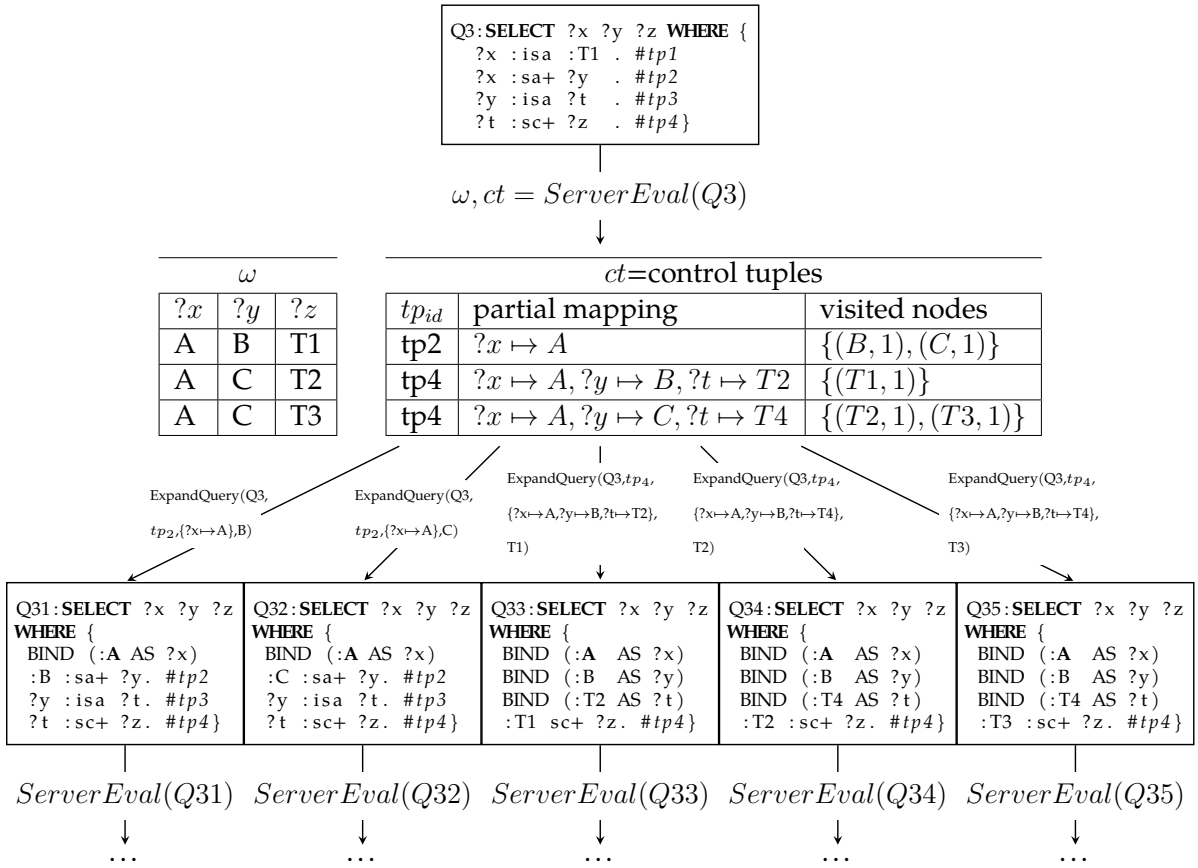


Figure 6.3 – First iteration of ClientEval($Q3$) as defined in Algorithm 7 with graph D and $MaxDepth = 1$.

6.3.3 The PTC-Client

The general idea of the PTC-Client is to use the control tuples returned by the $pPTC$ iterators to expand frontier nodes until no more frontier nodes can be discovered, i.e., transitive closures are complete.

Algorithm 7 describes the behavior of the PTC-Client. It is fundamentally an iterative Breadth-First Search (BFS) algorithm that traverses frontier nodes. The FIFO queue stores the frontier nodes to expand alongside their context. R stores the solutions mappings of the query. The V variable represents the visited nodes. As a property path pattern may be instantiated many times, each instance corresponding to the computation of a different transitive closure, we store a different set of visited nodes for each instance of a property path pattern, i.e., for each pair (ptc_{id}, μ) in control tuples.

Figure 6.3 illustrates the first iteration of Algorithm 7 using Query $Q3$ of Figure 6.1b.

First, $Q3$ is evaluated on the server by calling $ServerEval$ (Line 6). $ServerEval$ accepts any SPARQL property path query and returns a set ω of solutions mappings and a set ct of control tuples. The sets ω and ct for Query $Q3$ are depicted in Figure 6.3 by the two tables. As we can see, all visited nodes are discovered with a depth of 1. As $MaxDeph = 1$, all visited nodes are frontier nodes. Consequently, Algorithm 7 will expand $Q3$ with all these frontiers nodes.

$ExpandQuery$ takes a query Q, a set of partial mappings and a frontier node n as input (Line 16), and returns a new query Q' as output. $ExpandQuery$ processes in three steps. (1) The subject of the property path pattern identified by tp_{id} in Q is replaced by n in Q'. (2) Each triple (resp. property path) pattern $tp \in Q$ such that $\mu(tp)$ is not fully bounded is added to Q'. (3) To preserve the mappings μ from Q to Q', a BIND clause is created in Q' for each variable in $dom(\mu)$.

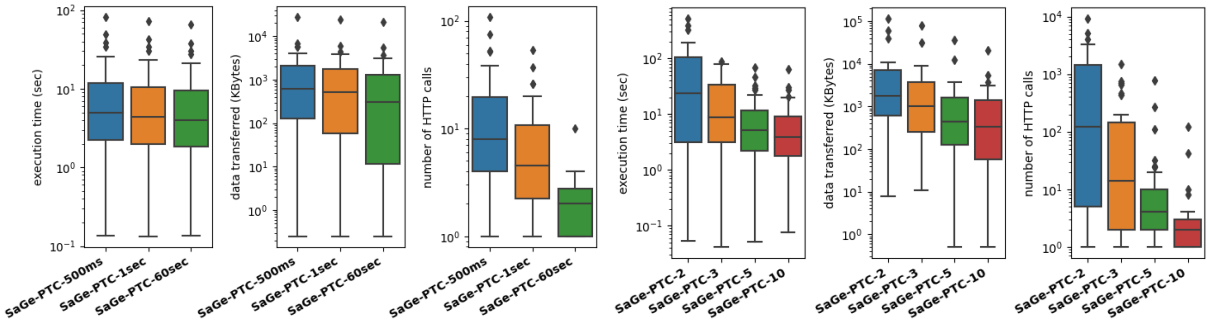
Figure 6.3 illustrates the query returned by $ExpandQuery$ for each frontier node returned by $ServerEval(Q3)$. Queries $Q31$, $Q32$, $Q33$, $Q34$ and $Q35$ are finally pushed in the FIFO queue (Line 17) to be evaluated at the next iteration. It could happen that the evaluation of an expanded query reaches an enqueued frontier node with a shortest path (Line 12). In this case, the expanded query is not evaluated (Line 5).

6.4 Experimental Study

In this experimental study, we want to empirically answer the following questions: What is the impact of $maxDepth$ and the time quantum parameters on the evaluation of SPARQL property path queries, both in terms of data transfer, number of HTTP calls and query execution time? How does the PTC approach perform compared to existing fair query services and SPARQL endpoints?

In our experiments *Jena-Fuseki* and *Virtuoso* are used as the baselines to compare our approach with SPARQL endpoints, while the multi-predicate automaton approach [7] is used as the baseline for the fair query services. We implemented the PTC operator in Python as an extension of the SaGe server, i.e., an open source implementation of Web Preemption, while the PTC-Client is implemented in JavaScript. The resulting system, i.e., the SaGe server with our PTC operator and the JavaScript Web client is called *SaGe-PTC*. The code and the experimental setup are available on the companion website [2].

2. https://github.com/JulienDavat/property-paths-experiments



(a) Impact of the quantum on performance

(b) Impact of the $maxDepth$ parameter on performance

Figure 6.4 – Impact of the different parameters on performance for the gMark queries.

6.4.1 Experimental Setup

Dataset and Queries: The dataset and queries are generated by gMark [25], a framework designed to generate synthetic graph instances coupled with complex property path query workloads. We use the "Shop" use case configuration file [3] to generate a graph instance of 7,533,145 triples and a workload of 30 queries. All our queries contain from 1 to 4 transitive closure expressions, for which numerical occurrences indicators have been replaced by Kleene plus "+" operators.

Compared Approaches: We compare the following approaches:

- *SaGe-PTC* is our implementation of the PTC approach. The dataset generated by gMark is stored using the SaGe HDT backend. The SaGe server is configured with a page size limit of 10000 solutions mappings and 10000 control tuples. Different configurations of *SaGe-PTC* are used. (i) *SaGe-PTC-2*, *SaGe-PTC-3*, *SaGe-PTC-5*, *SaGe-PTC-10* and *SaGe-PTC-20* are configured with a time quantum of 60 seconds and a $maxDepth$ of 2, 3, 5, 10 and 20, respectively. (ii) *SaGe-PTC-500ms*, *SaGe-PTC-1sec* and *SaGe-PTC-60sec* are configured with a $maxDepth$ of 20 and a time quantum of 500ms, 1sec and 60sec, respectively.
- *SaGe-Multi* is our baseline for the fair query service approaches. Property path queries are evaluated on the Web client using the multi-automaton decomposition-based approach [7]. For a fair evaluation, *SaGe-Multi* runs against the *SaGe-PTC* server with a time quantum of 60 seconds. We did not include Comunica [126] in the setup as

3. `https://github.com/gbagan/gmark/blob/master/use-cases/shop.xml`

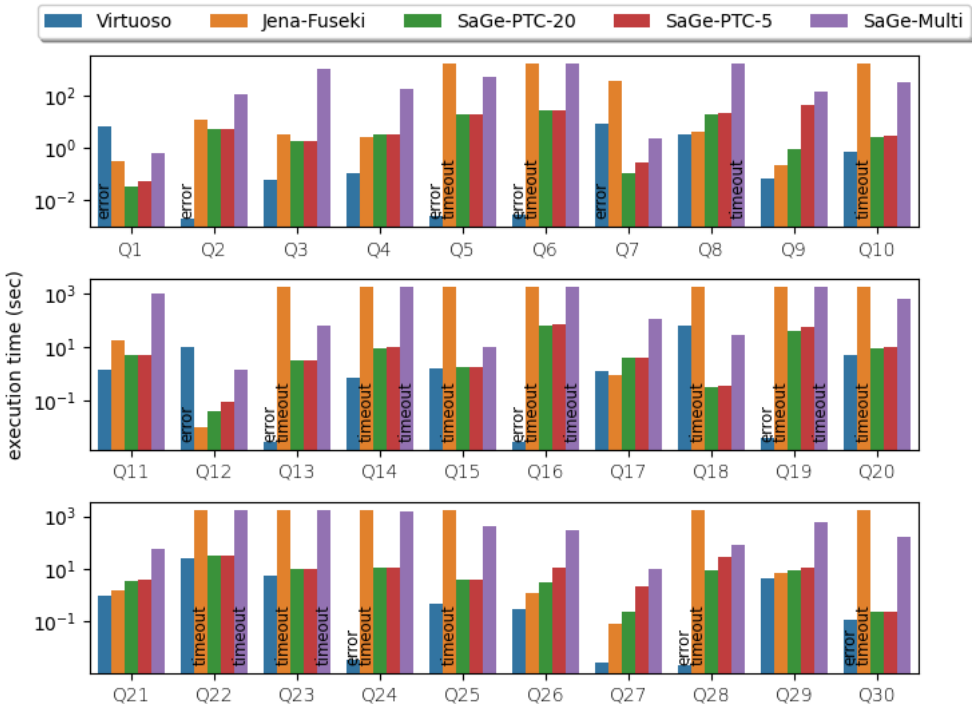


Figure 6.5 – Execution time per query for SPARQL endpoint and fair query service approaches compared to the *PTC* approach.

it has already been shown that *SaGe-Multi* dominates Comunica for all evaluation metrics [7].

- *Virtuoso* is the Virtuoso SPARQL endpoint (v7.2.5). *Virtuoso* is configured **without quotas** in order to deliver complete results. We also configured *Virtuoso* with a single thread to fairly compare with other engines.
- *Jena-Fuseki* is the Apache Jena Fuseki endpoint (v3.17.0) with the same configuration as *Virtuoso*, i.e., without quotas and a single thread.

Evaluation Metrics: Presented results correspond to the average obtained of three successive executions of the queries workload. Each query is evaluated with a timeout of 30 minutes. (1) *Execution time* is the total time between starting the query execution and the production of the final results by the Web client. (2) *Data transfer* is the total number of bytes transferred to the Web client during query execution. (3) *Number of HTTP calls* is the total number of HTTP calls issued by the Web client during query execution.

Hardware Setup: We run our experiments on a Google cloud virtual machine (VM) instance. The VM is a c2-standard-4 machine with 4 virtual CPU, 16GB of RAM and a 256GB SSD. Both clients and servers run on the same machine. Each Web client is instrumented to count the number of HTTP requests sent to the server, and the size of the data transferred from the server.

6.4.2 Experimental Results

What is the impact of the quantum on performance? To measure the impact of the quantum on performance, we run our workload with different quanta; 500ms, 1sec and 60sec. The $maxDepth$ is set to 20 for each quantum, such that all queries terminate without frontier nodes. Figure 6.4a presents *SaGe-PTC* performance for *SaGe-PTC-500ms*, *SaGe-PTC-1sec* and *SaGe-PTC-60sec*.

As expected, increasing the quantum improves performance. With large quantum, a query needs fewer calls to complete, which in turn improves query execution time. Concerning the data transfer, the visited nodes of the $pPTC$ iterators are flushed at the end of each quantum, which leads to revisit already visited nodes. Consequently, the less a query needs quanta to complete, the less it transfers duplicates.

What is the impact of maxDepth on performance? To measure the impact of the $maxDepth$ parameter on performance, we run our 30 queries with different settings for $maxDepth$; 2, 3, 5 and 10. To reduce the impact of the quantum, we choose a large quantum of 60 seconds. Figure 6.4b presents *SaGe-PTC* performance for *SaGe-PTC-2*, *SaGe-PTC-3*, *SaGe-PTC-5* and *SaGe-PTC-10*.

As we can see, $maxDepth$ impacts significantly the performance in terms of execution time, data transfer and number of HTTP calls. Increasing $maxDepth$ drastically improves performance as it allows to capture larger transitive closures. As a result, less control tuples are transferred to the Web client and less expanded queries are sent to the server.

How does the PTC approach perform compared to fair query services and SPARQL endpoints? The PTC approach computes SPARQL property path queries without joins on the Web client. When $maxDepth$ is high enough, no expanded queries are sent to the server. We just have to pay for the Web Preemption overheads and the duplicates transferred by the PTC approach. Consequently, we expect our approach to be

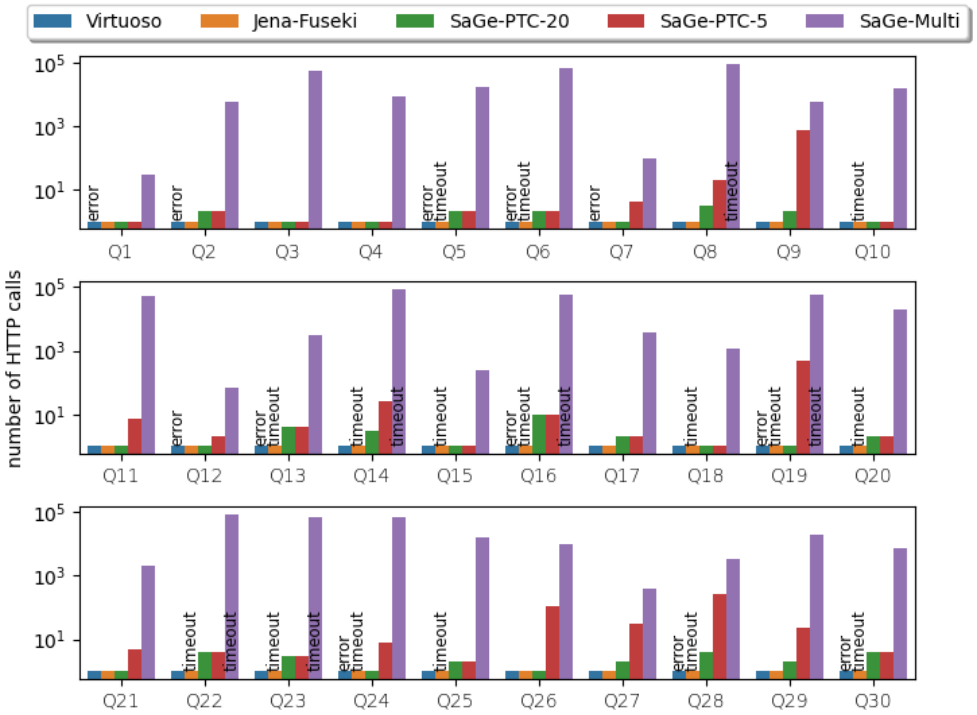


Figure 6.6 – Number of HTTP calls per query for SPARQL endpoint and fair query service approaches compared to the PTC approach.

somewhere between SPARQL endpoints and existing fair query services in terms of performance. Close to SPARQL endpoints when $maxDepth$ is high and better than existing fair query services in the general case. We compare *SaGe-PTC* with *SaGe-Multi*, *Jena-Fuseki* and *Virtuoso*. We run *SaGe-PTC* with both $maxDepth = 5$ (*SaGe-PTC-5*) and $maxDepth = 20$ (*SaGe-PTC-20*). Figures 6.5, 6.7 and 6.6 respectively present the execution time, the data transfer and the number of HTTP calls for each approach.

As expected, *SaGe-PTC* outperforms *SaGe-Multi* regardless the query. Because *SaGe-PTC* does not decompose BGPs, the number of HTTP calls is drastically reduced, as shown in Figure 6.6. Concerning the data transfer, both *SaGe-PTC* and *SaGe-Multi* transfer the visited nodes. However, *SaGe-PTC* does not transfer any intermediate results, saving a lot of data transfer.

Compared to *Jena-Fuseki* both approaches use a similar graph traversal algorithm. However, *Jena-Fuseki* has no overheads, i.e., *Jena-Fuseki* is optimal in terms of data transfer and number of HTTP calls. Only one call is sent per query and only the final results are transferred to the client. Consequently, we expect *Jena-Fuseki* to perform better than *SaGe-PTC*. Surprisingly *Jena-Fuseki* does not dominate the PTC approach. *SaGe-PTC-20*

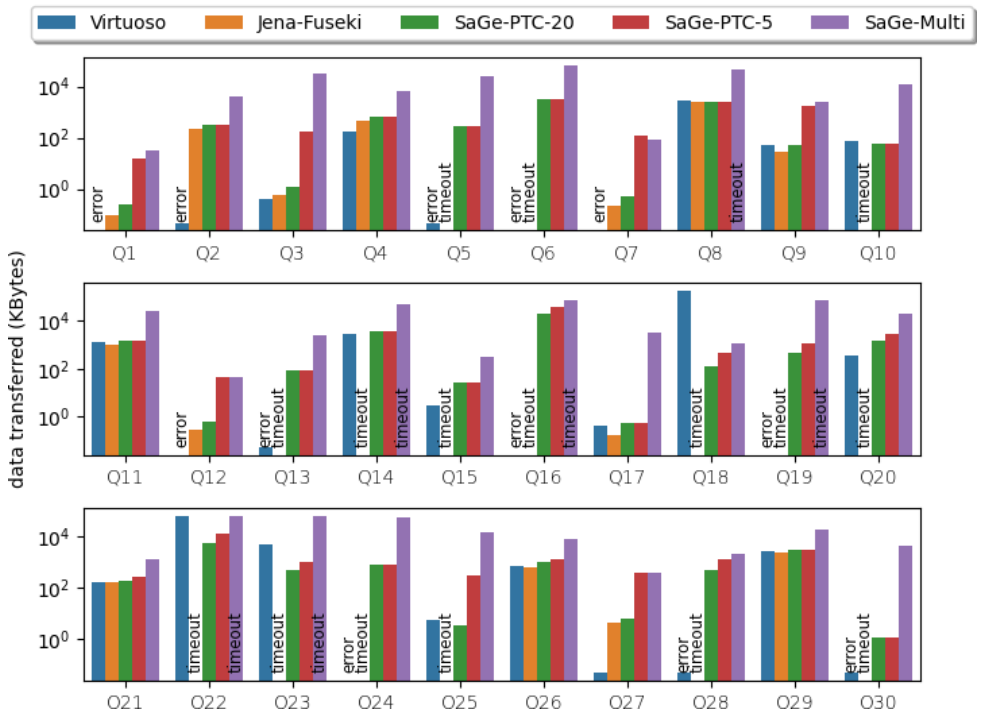


Figure 6.7 – Data transfer per query for SPARQL endpoint and fair query service approaches compared to the PTC approach.

is very close to *Jena-Fuseki* for queries where *Jena-Fuseki* does not time out. As queries return no frontier nodes, overheads compared to *Jena-Fuseki* are small. The differences between the two approaches are mainly due to a join ordering issue. Compared to *SaGe-PTC-20*, with *SaGe-PTC-5* queries need to send expanded queries to terminate. As expected, *SaGe-PTC-5* offers performance between that of SPARQL endpoints and that of existing fair query services. For most queries, its performance are very close to *SaGe-PTC-20*. We conjecture that most transitive closures for our queries can be computed with small $maxDepth$. When there is a large number of frontier nodes to explore, performance degrades but remains better than those of existing fair query services.

We expect *Virtuoso* to be optimal as it implements state-of-the-art query optimization techniques. Surprisingly, *Virtuoso* does not dominate *SaGe-PTC*. *Virtuoso* generates errors for 12 queries out of 30. It cannot execute the 12 queries either because of the missing of a starting point or because it has not enough space resources to materialize the transitive closure. *Virtuoso* issues are mainly due to the simple path semantics when dealing with dense graphs. Of course, the number of HTTP calls for *Virtuoso* is optimal. However, using a simple path semantics significantly impacts data transfer and query

execution time when property path queries are executed against dense graphs.

6.5 Conclusion

In this chapter, we have proposed an original approach to process SPARQL property path queries online and get complete results. Thanks to a preemptable Partial Transitive Closure operator, a Web client is ensured to grab all mappings that are reachable at a depth fixed by the server. Thanks to control information delivered during SPARQL property path queries processing, a Web client is able to generate new queries to find the missing mappings. Unlike current fair query service approaches, the PTC approach does not break BGPs containing property path patterns. Even in the presence of property path patterns, all joins are performed on the server without transferring intermediate results to the Web client. As demonstrated in the experiments, the PTC approach outperforms existing fair query service approaches and significantly reduces the gap in performance with SPARQL endpoints.

The PTC approach raises several interesting perspectives. First, there is a large room for optimization. How to find better join orders in the presence of property path patterns? How to improve the evaluation of property path patterns when they are "reachability" oriented? How to support nested transitive closures on the server? Second, it may be interesting to explore if Partial Transitive Closures and partial aggregations [53] are compatible. If the aggregation functions are computed on the Web client, then there is no issue for computing aggregations in the presence of property path patterns. However, evaluating both partial aggregations and Partial Transitive Closures on the server may return incorrect results.

Chapter 7

Join Ordering of SPARQL Property Path Queries

7.1 Introduction

In the previous chapters, we focused on pushing as much query processing load as possible on the server while ensuring fair query processing. We added an approximate COUNT-DISTINCT operator to reduce SPARQL aggregate queries data transfer, as well as a Partial Transitive Closure operator to improve execution performance of property path queries.

However, Leis et al. [84] demonstrated that good join orders are crucial to expect good query execution performance. To illustrate, Query $Q1$ presented in Figure 7.1a searches for road bicycle races in Central America on Wikidata. Q_1 has been executed on the Wikidata query service using two different join orders: (1) a join order $J_1 = ((tp1 \bowtie tp2) \bowtie tp_3)$ that has been decided by Blazegraph, the SPARQL engine behind Wikidata (2) a join order $J_2 = ((tp_3 \bowtie tp2) \bowtie tp1)$ that has been hand-crafted. Following J_1 the query $Q_1^{J_1}$ times out on Wikidata, i.e., $Q_1^{J_1}$ requires more than 60 seconds to complete, while $Q_1^{J_2}$ terminates in less than 3 seconds. Although the join ordering problem has been extensively studied in the context of conjunctive queries with filters [84, 73, 56], it has been poorly explored when considering property path queries [55, 54]. It is currently unclear how current engines consider property path patterns, i.e., how the cost of a join order that contains PPPs is computed, and how it should be computed.

Approach and contributions: In this work, we focus on the class of conjunctive two-way regular path queries with UNION and FILTER, denoted $C2RPQ_{UF}$ [34]. For $C2RPQ_{UF}$ queries, we propose a query optimizer that can find efficient join orders without changing existing SPARQL engines. Finding such join orders is challenging; depending on the join order, a property path pattern may behave as a transitive closure or a reachability pattern. The changing nature of property path patterns is biasing traditional cost mod-

```
SELECT ?x1 ?x3 WHERE {
    ?x3 wdt:P361 wd:Q27611 .   # tp1 (9)
    ?x1 wdt:P17 ?x3 .          # tp2 (13M)
    ?x1 wdt:P641* wd:Q3609 .   # tp3 (47K)
}
```

(a) $Q_1^{J_1}$: Blazegraph's join order

```
SELECT ?x1 ?x3 WHERE {
    hint:Query hint:optimizer "None" .
    ?x1 wdt:P641* wd:Q3609 . # tp3
    hint:Prior hint:gearing "reverse" .
    ?x1 wdt:P17 ?x3 .          # tp2
    ?x3 wdt:P361 wd:Q27611 . # tp1
}
```

(b) $Q_1^{J_2}$: Hand-crafted join order

Figure 7.1 – Query Q_1 comes from the Wikidata Query Benchmark [17] and returns road bicycle races located in Central America. $Q_1^{J_1}$ is the join order decided by Blazegraph while $Q_1^{J_2}$ is a hand-crafted join order. The "hint:Query" triple pattern in $Q_1^{J_2}$ is used to force the join order in Blazegraph. Commented numbers are triple patterns cardinality. On the Wikidata Query Service $Q_1^{J_1}$ times out (>60s) while $Q_1^{J_2}$ terminates in less than 3 seconds.

els that fail to find efficient join orders. We made the following contributions:

1. We propose a cost model along with a Dynamic Programming (DP) algorithm that is able to capture the cost of evaluating PPQs using traditional PPP operators. Compared to state-of-the-art, the proposed DP algorithm can rewrite PPPs such that their cost remains observable to existing cost functions such as the C_{out} [83].
2. Any cost model requires accurate cardinality estimates. However, there is currently no cardinality estimator able to handle property path patterns. To fill this gap, we propose a cardinality estimator for PPQs based on random walks. Random walks can be computed cheaply thanks to B-Tree indexes, widely used to index RDF data. Compared to state-of-the-art, the proposed cardinality estimator extends the WanderJoin approach [83] to handle property path patterns.
3. The approach is evaluated on Blazegraph and Virtuoso using the newly proposed Wikidata Query Benchmark [17]. Experimental results demonstrate that our approach significantly improves the execution time of property path queries. Compared to Blazegraph, the execution time is divided by at least 14.

The rest of the chapter is organized as follows: Section 7.2 presents our approach to find good join orders in the presence of property path patterns. Section 7.3 introduces a cost model for property path queries. Section 7.4 presents our cardinality estimator for property path queries. Section 7.5 details our experimental results. Finally, we present our conclusions and future work in Section 7.6.

7.2 Optimization of Property Path Queries

Our proposal follows the traditional query optimizer architecture of the system R [119]. The optimizer takes a property path query as input and returns a physical plan that minimizes a cost function. For each property path pattern, the optimizer also decides in which direction it should be evaluated, i.e., from subjects to objects (*forward navigation*) or from objects to subjects (*backward navigation*). The query optimizer enumerates valid join orders using a dynamic programming algorithm. Based on cardinality estimates, the cost model chooses the cheapest alternative among the valid join orders. The goal of this work is to target mainstream SPARQL engines, consequently two hypotheses are assumed: (1) there are no indexes dedicated to transitive closures, such as the FERRARI index [121]; (2) property path patterns are evaluated using the traditional ALP procedure (a BFS-style algorithm) defined in the SPARQL specification [61] as it is done in JENA or Blazegraph, or using transitive closure operators as in Virtuoso.

As defined in the previous chapter, a SPARQL Property Path Query (PPQ) is a SPARQL query with at least one property path pattern (PPP). A PPP is a tuple in $(I \cup B \cup V) \times E \times (I \cup B \cup L \cup V)$ where E is the set of property path expressions defined by the following grammar [81]: $e := a \mid e^- \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^+ \mid e^* \mid e? \mid !a_1, \ldots, a_k \mid !a_1^-, \ldots, a_k^-$ where $a, a_1, \ldots, a_k \in I$. This work assumes non-transitive property path expressions to be evaluated using traditional SPARQL algebra operators [61]. Thus, our proposal only evaluates transitive property path expressions, i.e., e^+ and e^*. Nested transitive closures such as $(ab+)+$, are not considered and will be the subject of future work.

The Join Ordering Problem with Property Paths

$$C_{mm}(P,G) = \begin{cases} |[\![P]\!]_G| & \textbf{if } P = tp \vee P = \sigma(tp) \\ C_{mm}(P_1) + & \textbf{if } P = P_1 \overset{NLJ}{\bowtie} P_2, \\ \quad |[\![P_1]\!]_G| \times max(\frac{|[\![P_1 \bowtie tp]\!]_G|}{|[\![P_1]\!]_G|}, 1) & (P_2 = tp \vee P_2 = \sigma(tp)) \end{cases}$$

Let us consider the C_{mm} cost function defined above, which is a simplified version of the one presented by Leis et al. [84]. For simplicity, only index-nested-loop joins and left-deep trees are considered, but our proposal holds in the general case. Most cost functions rely on cardinality estimates. For instance, the C_{mm} function defines the

cost of evaluating a triple pattern as its cardinality [84]. Indeed, considering traditional indexes SPO, POS, OSP as available, any triple pattern tp can be evaluated over an RDF graph G in $\mathcal{O}(|[\![tp]\!]_G| \log(|G|))$. Thus, for conjunctive queries with filters, minimizing a cardinality-based cost function such as the C_{mm} effectively leads to good join orders [41, 84]. While the cost of evaluating a triple pattern is correlated with its cardinality, it is not always true for property path patterns (PPPs). Assuming PPPs are evaluated using ALP, a BFS-style algorithm defined by the standard [61], two cases can be distinguished:

Transitive-pattern Let $tp = (s, p, o)$ be a PPP such that at least the subject or the object is a variable, i.e., $s \in V \vee o \in V$. Let o be in V and N be the set of nodes reachable from s. Using the ALP algorithm, the evaluation of tp returns N that can be computed in $\mathcal{O}(|N|log(|G|))$ over an RDF graph G. As $|[\![tp]\!]_G| = |N|$, the cost of evaluating tp is correlated with its cardinality.

Reachability-pattern Let $tp = (s, p, o)$ be a PPP such that both the subject and the object are bounded, i.e., $s, o \notin V$. According to [61], the cardinality of a fully bounded PPP is 1 if o can be reached from s, 0 otherwise. However, to check the reachability between s and o, the ALP algorithm first computes the set of nodes N reachable from s, then checks if $o \in N$. Consequently, the cost of evaluating a fully bounded PPP is not correlated with its cardinality.

To illustrate the problem on a concrete example, let us compute the cost of $Q_1^{J_1}$ and $Q_1^{J_2}$ depicted in Figure 7.1. Using the true cardinalities [1], we calculate $C_{mm}(Q_1^{J_1}) \approx 91K$ and $C_{mm}(Q_1^{J_2}) \approx 168K$. Despite $Q_1^{J_1}$ being estimated less costly than $Q_1^{J_2}$, $Q_1^{J_1}$ times out on Wikidata while $Q_1^{J_2}$ completes in less than 3 seconds. Focusing on property path patterns, tp_3 appears as a transitive-pattern in $Q_1^{J_2}$, while it appears as a reachability-pattern in $Q_1^{J_1}$. Because the cost of computing N is not captured by the cardinality of a reachability-pattern, the cost of $Q_1^{J_1}$ is largely underestimated. Thus, a cost function purely based on cardinalities, such as the C_{mm}, cannot correctly estimate the cost of a PPQ. *The scientific challenge is to define a cost model that, given any join order, can capture the cost of a PPP, whether it appears as a transitive-pattern or a reachability-pattern.*

```
SELECT ?x1 ?x3 WHERE {
    ?x3 wdt:P361 wd:Q27611 .      # tp1 (9)
    ?x1 wdt:P17 ?x3 .             # tp2 (13M)
    ?x1 wdt:P641* ?relax .
    FILTER (?relax = wd:Q3609) . # tp3 (47K)
}
```

(a) $Q_1^{J_1^F}$: Forward relaxation of J_1.

```
SELECT ?x1 ?x3 WHERE {
    ?x3 wdt:P361 wd:Q27611 .      # tp1 (9)
    ?x1 wdt:P17 ?x3 .             # tp2 (13M)
    ?relax wdt:P641* wd:Q3609 .  # tp3 (47K)
    FILTER (?relax = ?x1) .
}
```

(b) $Q_1^{J_1^B}$: Backward relaxation of J_1.

Figure 7.2 – Forward and Backward relaxations of tp_3 in J_1 of Query Q_1.

7.3 Cost-model for Property Path Queries

Given a join order J, the key idea is to relax fully bounded property path patterns (PPPs) such that J no longer contains reachability-patterns. If all PPPs behave as transitive-patterns, then cardinality-based cost functions are able to correctly estimate the cost of J. Whether a PPP behaves as a transitive-pattern or a reachability-pattern depends on the join order. Therefore, the general approach is to detect reachability-patterns during enumeration of join orders, and relax them before ordering them.

Definition 12 (Reachability-pattern relaxation) *Let Q be a SPARQL property path query, J a join order, and $tp = (s, p, o) \notin J$ a fully bounded property path pattern with respect to J, i.e., $var(tp) \subseteq var(J)$. A forward relaxation of tp generates a filter graph pattern of the form $((s, p, v)\ FILTER\ (v = o))$ such that $v \notin var(Q)$. A backward relaxation of tp generates a filter graph pattern of the form $((v, p, o)\ FILTER\ (v = s))$ such that $v \notin var(Q)$.*

Using a BFS-style algorithm, a reachability-pattern can be evaluated following two strategies. One can decide to navigate from the subject to the object (forward strategy), another can decide to go from the object to the subject (backward strategy). In this context, the forward and backward relaxations allow to estimate which strategy is the cheapest one. As the cost of going forward or backward can be drastically different [145], selecting the best strategy is crucial to expect good performance. For instance, Blazegraph evaluates tp_3 in $Q_1^{J_1}$ starting from the object. Using the forward and backward relaxations, we can rewrite $Q_1^{J_1}$ as $Q_1^{J_1^F}$ and $Q_1^{J_1^B}$ as depicted in Figure 7.2. If we use the true cardinalities to compute the cost of both strategies, we get $C_{mm}(Q_1^{J_1^F}) = 95K$ and $C_{mm}(Q_1^{J_1^B}) = 2.6B$. Thus, $Q_1^{J_1}$ times out on Wikidata because Blazegraph chose the wrong strategy to evaluate tp_3.

1. True cardinalities were computed using SPARQL COUNT queries on the Wikidata SPARQL endpoint as of December 5, 2022

Algorithm 8: Dynamic programming algorithm with relaxation

Require: Q: SPARQL Property Path Query, G: RDF Graph
Data: $dpTable$: keeps the best join order for a set S of triple/property path patterns

```
1  for tp ∈ Q do dpTable[{tp}] = tp
2  for n ∈ 1..|Q| − 1 do
3  |  for S ∈ dpTable : |S| = n do
4  |  |  for tp ∈ Q : tp ∉ S do
5  |  |  |  if var(tp) ∩ var(dpTable[S]) = ∅ then continue
6  |  |  |  S' = S ∪ {tp} ; C = ∅
7  |  |  |  if tp is a PPP ∧ var(tp) ⊆ var(dpTable[S]) then
8  |  |  |  |  C = C ∪ {dpTable[S] ⋈ ForwardRelaxation(Q, tp)}
9  |  |  |  |  C = C ∪ {dpTable[S] ⋈ BackwardRelaxation(Q, tp)}
10 |  |  |  else
11 |  |  |  |  C = C ∪ {dpTable[S] ⋈ tp}
12 |  |  |  for P ∈ C do
13 |  |  |  |  if S' ∉ dpTable ∨ C_mm(P, G) < C_mm(dpTable[S'], G) then
14 |  |  |  |  |  dpTable[S'] = P
15 return UndoRelaxation(dpTable[Q])
```

Algorithm 8 is a custom dynamic programming algorithm that integrates relaxation. When the algorithm detects a reachability-pattern tp with respect to a join order $J = dpTable[S]$ (Line 7), it uses relaxation so that the cost function is able to correctly estimate the cost of tp in J. Moreover, to select the best strategy to evaluate tp both relaxations are used, generating two candidates that are stored in C (Lines 8-9). One of them will evaluate tp using the forward strategy, while the other will use the backward strategy. Next, the cost function is used to keep the cheapest alternative (Lines 12-14). At the end, the algorithm returns the cheapest join order, with relaxed property path patterns in their original form.

7.4 Cardinality Estimation of Property Path Queries

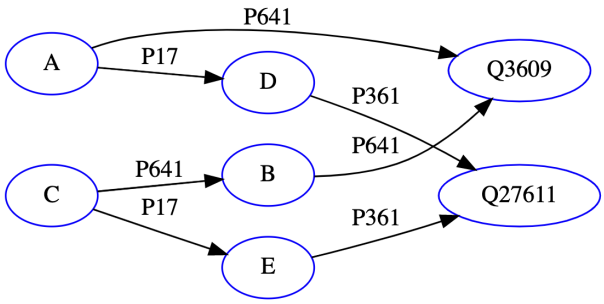
This section introduces a new cardinality estimator for property path queries based on random walks. Random walks [85] offer several advantages: (1) they proved to be the best approach for estimating the cardinality of conjunctive SPARQL queries [99]; (2) they do not require maintaining statistics [54]; (3) they can be efficiently imple-

```
SELECT DISTINCT ?x1 ?x3 WHERE {
    ?x1 wdt:P641 wd:Q3609 .   # tp3
    ?x1 wdt:P17 ?x3 .         # tp2
    ?x3 wdt:P361 wd:Q27611 .  # tp1
}
```

(a) $(Q_1^{J_2})^{1..1}$

```
SELECT DISTINCT ?x1 ?x3 WHERE {{
    ?x1 wdt:P641 wd:Q3609 .   # tp3
    ?x1 wdt:P17 ?x3 .         # tp2
    ?x3 wdt:P361 wd:Q27611 .  # tp1
} UNION {
    ?v1 wdt:P641 wd:Q3609 .   # tp3.1
    ?x1 wdt:P641 ?v1 .        # tp3.2
    ?x1 wdt:P17 ?x3 .         # tp2
    ?x3 wdt:P361 wd:Q27611 .  # tp1
}}
```

(b) $(Q_1^{J_2})^{1..2}$



(c) RDF graph G_1

Figure 7.3 – RDF graph G_1 and rewrites of the query $Q_1^{J_2}$ used to estimate the cost of J_2 with random walks.

mented just by relying on traditional SPO, POS, and OSP indexes that are widely available on existing triple stores [83]. Before moving to the contribution, we first recall how to estimate the cardinality of conjunctive SPARQL queries using random walks. Next, we address the case of SPARQL property path queries. For the sake of simplicity, we assume that property path queries are conjunctive queries with a single property path pattern. However, the approach can be generalized to $C2RPQ_{UF}$ queries that contain multiple property path patterns.

7.4.1 Cardinality Estimates of Conjunctive Queries

Let Q be a conjunctive SPARQL query, and $J = \langle tp_1, \ldots, tp_n \rangle$ be the join order used to perform random walks. Based on WanderJoin [85] a random walk $\gamma = \langle t_1, \ldots, t_n \rangle$ is computed over an RDF graph G by randomly picking t_1 in $[\![tp_1]\!]_G$, and each subsequent t_i $(i > 1)$ in $[\![t_{i-1} \bowtie tp_i]\!]_G$. Thus, the probability of sampling γ is $P(\gamma) = |[\![tp_1]\!]_G|^{-1} \prod_{i=2}^{n} |[\![t_{i-1} \bowtie tp_i]\!]_G|^{-1}$. Let $\Gamma = \langle \gamma_1, \ldots, \gamma_k \rangle$ be a multiset of k random walks, the cardinality of Q is estimated as $card(\Gamma) = |\Gamma|^{-1} \sum_{i=1}^{|\Gamma|} P(\gamma_i)^{-1}$. For instance, let us estimate the cardinality of $(Q_2^{J_2})^{1..1}$ on the RDF graph G_1 with a budget of 2 random walks. Both $(Q_2^{J_2})^{1..1}$ and G_1 are depicted in Figure 7.3. Let γ_1 and γ_2 be the two random walks we picked following J_2:

	tp_3	picking t_1	$= (\mathbf{A}, P641, Q3609)$	in $[\![(?x1, P641, Q3609)]\!]_{G_1}$
γ_1	tp_2	picking t_2	$= (A, P17, \mathbf{D})$	in $[\![(\mathbf{A}, P17, ?x3)]\!]_{G_1}$
	tp_1	picking t_3	$= (D, P361, Q27611)$	in $[\![(\mathbf{D}, P361, Q27611)]\!]_{G_1}$
γ_2	tp_3	picking t_1	$= (\mathbf{B}, P641, Q3609)$	in $[\![(?x1, P641, Q3609)]\!]_{G_1}$

In this example, $P(\gamma_1) = \frac{1}{2} \times \frac{1}{1} \times \frac{1}{1} = \frac{1}{2}$, while $P(\gamma_2) = 0$. Indeed, when it becomes impossible for a random walk γ to sample t_i for some $i \leq n$, e.g., t_2 in γ_2 because $[\![(B, P17, ?x3)]\!]_{G_1} = \emptyset$, γ is classified as invalid, and its probability of being sampled is 0. Thus, the estimated cardinality of $(Q_2^{J_2})^{1..1}$ is $\frac{1}{2} \times (P(\gamma_1)^{-1} + P(\gamma_2)^{-1}) = \frac{2+0}{2} = 1$.

7.4.2 Cardinality Estimates of Property Path Queries

Definition 13 *Let Q be a conjunctive SPARQL property path query. Let $tp_i \in Q$ be a property path pattern. We denote Q^d the conjunctive SPARQL query obtained by rewriting tp_i into a chain $tp_i^1, \ldots, tp_i^d$ of triple patterns. If $J = \langle tp_1, \ldots, tp_i, \ldots, tp_{|Q|} \rangle$ is a join order associated to Q, we denote $J^d = \langle tp_1, \ldots, tp_i^1, \ldots, tp_i^d, \ldots, tp_{|Q|} \rangle$ the equivalent join order associated to Q^d.*

To estimate the cardinality of a SPARQL property path query Q, the key idea is to rewrite Q into an equivalent query Q' that does not contain property paths. For instance, let us consider the query $Q_1^{J_2}$ depicted in Figure 7.1b. Assuming that the diameter d of the relation $P641$ is known, and let $d = 2$, $Q_1^{J_2}$ is equivalent to the query $(Q_1^{J_2})^{1..2}$ described in Figure 7.3b, i.e., both return the same result. Knowing d, any PPP can be rewritten as an `UNION` graph pattern with d clauses, each clause matching paths of different lengths from 1 to d. Thus, assuming a budget of k random walks, the cardinality of Q' can be estimated by uniformly distributing random walks over the d clauses of the `UNION`, ending up with d multisets of random walks $\Gamma_1, \ldots, \Gamma_d$. The cardinality of Q' is then estimated as $\sum_{i=1}^{d} card(\Gamma_i)$. In other words, we consider clauses of the `UNION` as individual queries $Q^1, \ldots, Q^d$, for which we estimate the cardinality, and the cardinality of Q is the sum of the estimated cardinalities of $Q^1, \ldots, Q^d$.

According to the SPARQL semantics, PPPs are evaluated following a set semantics [61]. For instance, no matter how many paths they are between `wd:Q3609` and `A`, evaluating tp_3 over G_1 must return `A` only once. Thus, for the rewriting to be correct, a `DISTINCT` modifier must be introduced in the rewriting of Q into Q'. However, to the best of our knowledge, estimating the cardinality of `DISTINCT` queries using random walks has not been studied. To cope with this issue, the `DISTINCT` modifier is just ignored, aware that the estimator will overestimate cardinalities. On dense graphs, cardinalities

can be significantly overestimated, preventing the optimizer from finding good join orders. Nevertheless, we assume that in practice, removing the `DISTINCT` modifier will not prevent the optimizer from finding good join orders.

Algorithm 9: Cardinality estimation with property paths

Require: Q: SPARQL query where tp_i is a property path pattern, J: Join order, G: RDF graph, k: Number of random walks, $dMax$: Depth exploration limit

Data: d: Length of the longest path explored, Γ: Multisets of random walks

```
1  d = 1
2  while Σ_j |Γ_j| < k do
3  |  d' ~ U{1, d} ; γ = ⟨t_1, ..., t_n⟩ = randomWalk(Q^d', J^d', G)
4  |  if P(γ) > 0 ∧ t_i^1, ..., t_i^d' ∈ γ are pairwise distinct then
5  |  |  Γ_d' = Γ_d' ∪ {γ}
6  |  else
7  |  |  Γ_d' = Γ_d' ∪ {γ'} with P(γ') = 0
8  |  if |γ| ≥ i + d − 1 ∧ t_i^1, ..., t_i^d ∈ γ are pairwise distinct then
9  |  |  d = min(d + 1, dMax)
10 return Σ_{j=1}^{d} card(Γ_j)
```

Rewriting a property path pattern tp requires to know the diameter d of the subgraph recognized by tp. To avoid relying on statistics, Algorithm 9 computes d while performing random walks. Given a SPARQL property path query Q where tp_i is a PPP, and a budget k, Algorithm 9 starts with $d = 1$. At each iteration, the algorithm computes a random walk $\gamma = \langle t_1, \ldots, t_n \rangle$ from $Q^{d'}$, following the join order $J^{d'}$, where d' is drawn uniformly at random between 1 and d. Each time $d' = d$, the algorithm checks if γ has found a path of length d matching tp_i. Given $J^d = \langle tp_1, \ldots, tp_i^1, \ldots, tp_i^d, \ldots, tp_{|Q|} \rangle$, a path of length d has been found if γ matches at least $\langle tp_1, \ldots, tp_i^1, \ldots, tp_i^d \rangle$, i.e., if $|\gamma| \geq |\langle tp_1, \ldots, tp_i^1, \ldots, tp_i^d \rangle|$ or $|\gamma| \geq i + d - 1$. In this case, it may exist a path of length $d + 1$ matching tp_i, and d is increased by 1.

Algorithm 9 increases d each time a random walk finds a path of length d matching tp_i. However, in the presence of cycles, d may increase forever, significantly impacting the accuracy of estimates. To address this issue, Algorithm 9 enforces a simple path semantics. Under a simple path semantics [87], a random walk can go through a node only once when matching PPPs. In other words, let $\gamma = \langle t_1, \ldots, t_i^1, \ldots, t_i^{d'}, \ldots, t_{|Q|} \rangle$ be a random walk sampled from $Q^{d'}$, with $t_i^1, \ldots, t_i^{d'}$ matching $tp_i^{d'}$, γ is valid if and only if $t_i^1, \ldots, t_i^{d'}$ are pairwise distinct. Thus, considering an infinite number of random walks,

Table 7.1 – Characteristics of the workload used in the experiments.

#queries	#joins	#triples	#path patterns	#constants	#join variables
213	1 - 8	2 - 9	1 - 2	1 - 5	1 - 6

Algorithm 9 ensures that d converges to the size of the longest simple path in the subgraph recognized by tp_i, which is equal to or larger than the diameter of the subgraph.

Even without cycles, d can quickly reach large values. Because the budget k is distributed between queries $Q^1, \ldots, Q^d$ to sample paths of length 1 to d, with a small budget, the algorithm may end up with too few random walks in each multiset Γ_j to compute accurate estimates. To address this issue, d can be clipped to a maximum value $dMax$. As the computation time of a random walk is proportional to its size, clipping d can also improve optimization times.

7.5 Experimental Study

The goal of this experimental study is to empirically answer the following questions: (1) Does our approach improve total workload execution time compared to baselines? (2) What is the impact of our approach on each query? (3) How do the budget k and the depth exploration limit $dMax$ impact performance?

7.5.1 Experimental Setup

Datasets and Queries. Our experiments use the newly proposed Wikidata Graph Query Benchmark (WDBench) [17], which extracts real-world SPARQL queries from the public query logs of the Wikidata SPARQL endpoint. The WDBench provides a RDF dataset of 1,257,169,959 triples built from the dump of Wikidata. To create our workload, all non-property path queries have been filtered out, as well as queries with cross-products. The resulting workload contains 213 queries and is described in Table 7.1. To ensure that queries return the same result for all engines, we added the DISTINCT modifier to all queries.

Compared Approaches. To demonstrate that random walks have the potential to be used to improve the join order of SPARQL property path queries, we compare our approach with Virtuoso v.OS-7.2.7 [46] (one of the most deployed engine in practice [19],

as well as the SPARQL endpoint behind DBpedia), and Blazegraph v.2.1.4 [127] (used by the Wikidata Query Service [90]). Blazegraph comes with two different optimizers; a first optimizer based on simple statistics such as the cardinality of triple patterns, and another one, named RTO, that relies on sampling to estimate the cost of join orders. Because RTO supports property paths and is adapted from ROX [78], which is related to our proposal, our approach is compared to both optimizers.

Implementation and Experimental Protocol. We implemented our query optimizer as a standalone Python 3.9 program. Random walks are performed over the WDBench dataset stored in HDT [49]. The generated plans of our query optimizer are translated into SPARQL queries using Blazegraph and Virtuoso query hints. Query hints allow us to force the join order and the direction in which property path patterns are evaluated by the engine [2] [3] [4]. Virtuoso and Blazegraph have been tuned for a system with 64GB, following engine recommendations. Code and configurations can be found online for reproducibility purposes [5]. As random walks are not deterministic, the workload is optimized 5 times for each tested configuration, and each query is executed 3 times after a warmup execution. All queries are executed with a timeout of 900 seconds.

Evaluation Metrics. In our experiments, the following metrics are used: (1) The *Total Execution Time* is the time spent by Virtuoso or Blazegraph executing a SPARQL query with the Optimization Time. (2) The *Execution Time* is the time spent by Virtuoso or Blazegraph executing a SPARQL query without the Optimization Time. (3) The *Optimization Time* is the time spent by our query optimizer to optimize a SPARQL query. Each query is executed three times and the metrics are computed on the average of these 3 executions.

Hardware. All experiments ran on a single machine with Ubuntu 20.04.4 LTS, AMD EPYC 7513 32-Core Processor, 64GB of RAM, and a logical volume of 2TB on a remote SSD accessible through the LAN.

2. https://docs.openlinksw.com/virtuoso/rdfsparqlimplementatiotrans
3. https://docs.openlinksw.com/virtuoso/rdfperfcost
4. https://github.com/blazegraph/database/wiki/QueryHints
5. https://github.com/JulienDavat/Join-Ordering-of-SPARQL-Property-Path-Queries

Table 7.2 – TET=Total Execution Time, T=Timeouts, E=Errors.

Engine	Optimizer	TET [Seconds]	T	E
Blazegraph	Default	20270	15	9
	RTO	35107	36	64
	Proposal (k=1000, d=5)	1429 ± 130	0	0
Virtuoso	Default	419	0	0
	Proposal (k=1000, d=5)	362 ± 11	0	0

7.5.2 Experimental Results

Does our approach improve total workload execution time compared to baselines? Table 7.2 presents the total execution time of the workload for all engines. Our proposal is configured with a budget of 1000 random walks (k=1000), and the depth limit for property paths is set to 5 (d=5). As our proposal relies on random walks, the workload has been optimized 5 times. Averages and standard deviations are reported in Table 7.2.

First, Virtuoso is much faster than Blazegraph on this workload. While Blazegraph requires more than 20270 seconds, Virtuoso only needs 419 seconds. Moreover, Blazegraph is not able to execute 24 queries. For the first 15 queries, Blazegraph reaches the timeout set to 900 seconds in our experiments. An Out-Of-Memory exception occurs for the nine remaining queries. Note that queries that time out account for 900 seconds in the total execution time, while queries that result in an error account for 0. On its side, Virtuoso processes all 213 queries. Despite being recommended when there are join ordering issues, the RTO optimizer delivers the worst results. Given a SPARQL query Q, to estimate the cost of a join order J, RTO samples the first triple pattern in J and executes Q^J on this sample. Unfortunately, evaluating Q^J, even on a small sample, can take a very long time. As a result, many queries time out or crash because of the optimization step. In the end, our proposal outperforms both Blazegraph and Virtuoso. The workload total execution time is divided by at least 14 on Blazegraph. It can be much more depending on the queries that time out. Compared to Virtuoso, we observe a 14% improvement on the execution time.

What is the impact of our approach on each query? Figure 7.4 presents a per-query view of the results summarized in Table 7.2. Queries are ordered on the x-axis according to the total execution time of the baseline. Focusing first on Blazegraph, the 15 queries that exceed the time limit are depicted in dark gray, while the nine queries that result in an error are in red. As depicted by the blue curve in Figure 7.4, our optimizer makes

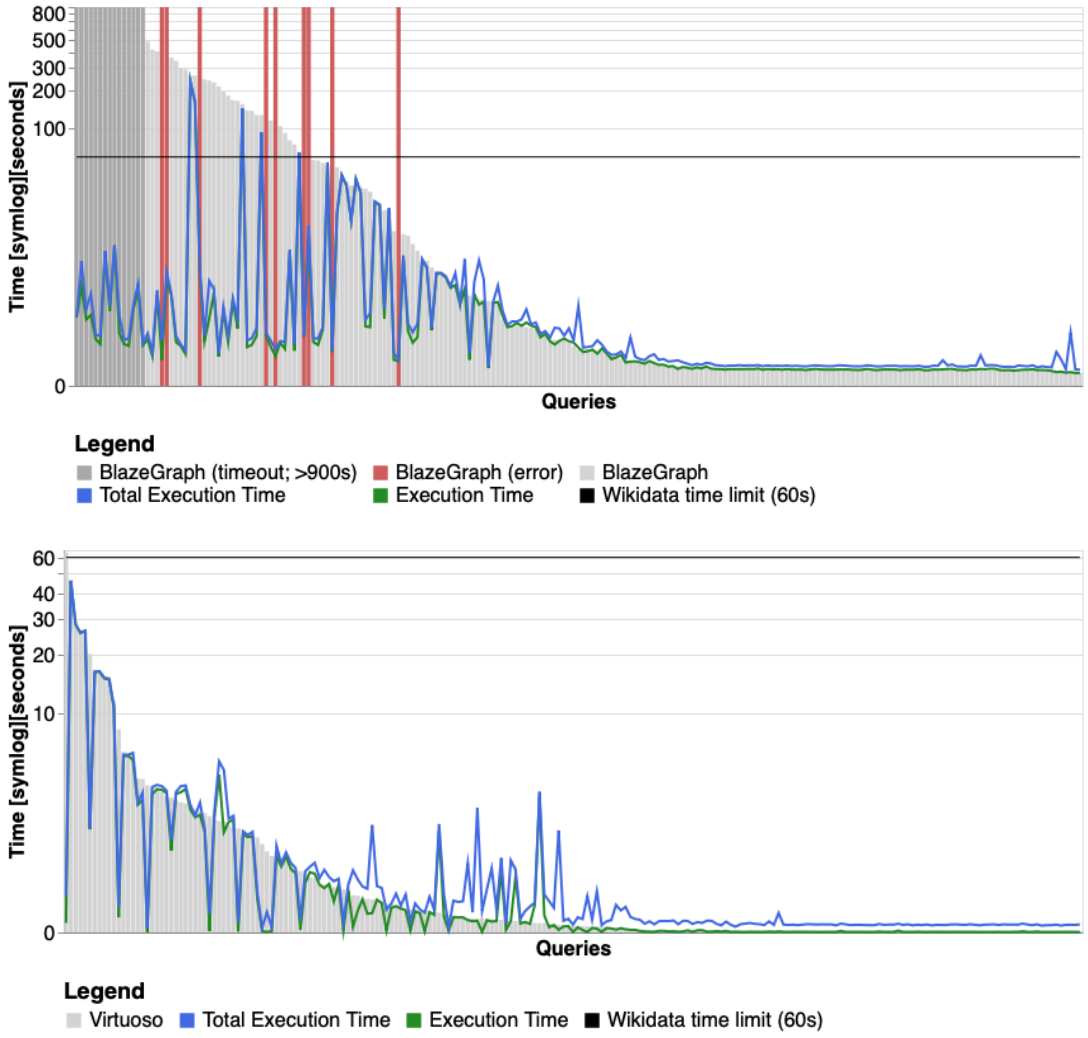


Figure 7.4 – Average execution time on Blazegraph (top) and Virtuoso (bottom) with a budget of 1000 random walks and an exploration depth limited to 5.

the difference on the long-running queries by finding better join orders. When looking at the optimization time, i.e., the ratio between the blue and green curves, they are irregularities. It comes from the HDT storage that cannot draw a random triple in constant time using the POS index. We can draw the same conclusion on Virtuoso. For short-running queries, the generated join orders are close to Virtuoso. However, long-running queries can benefit from significant improvements. For instance, the longest query on Virtuoso takes 63 seconds to complete. Using our optimizer we are able to find a join order that reduces the execution time to 110ms.

Table 7.3 – Global results of our experiments. For each configuration, queries have been optimized 5 times and executed 3 times after a warmup execution. DL=Depth Limit, TET=Total Execution Time (secs), ET=Execution Time (secs), OT=Optimization Time (secs), T=Timeouts, E=Errors.

		Walks	DL	TET	ET	OT	T	E
Blazegraph	Default			20270			15	9
	RTO			35107			36	64
	Proposal	1000	10	1635 (± 195)	1577 (± 195)	58 (± 2)	0	0
			5	1429 (± 130)	1383 (± 128)	37 (± 14)	0	0
			3	1629 (± 207)	1583 (± 206)	46 (± 4)	0	0
			1	2046 (± 219)	2006 (± 222)	40 (± 13)	0	0
		10000	10	1890 (± 286)	1530 (± 288)	361 (± 4)	0	0
			5	1545 (± 134)	1226 (± 127)	319 (± 14)	0	0
			3	1583 (± 206)	1290 (± 206)	293 (± 8)	0	0
			1	2087 (± 15)	1847 (± 11)	241 (± 6)	0	0
Virtuoso	Default			419	415	4	0	0
	Proposal	1000	10	365 (± 18)	317 (± 18)	48 (± 0)	0	0
			5	362 (± 11)	319 (± 11)	43 (± 0)	0	0
			3	356 (± 29)	317 (± 29)	39 (± 0)	0	0
			1	346 (± 5)	315 (± 5)	31 (± 0)	0	0
		10000	10	613 (± 17)	300 (± 12)	314 (± 9)	0	0
			5	594 (± 15)	303 (± 15)	291 (± 1)	0	0
			3	570 (± 16)	301 (± 35)	269 (± 1)	0	0
			1	528 (± 3)	308 (± 3)	220 (± 0)	0	0

How do the budget k and the depth exploration limit $dMax$ impact performance? In our approach, two parameters impact performance; the budget k, i.e., the number of random walks used to estimate the cardinality of joins, and the limit $dMax$ on the exploration depth for property paths. To measure the impact of these two parameters, we tested different configurations that are summarized in Table 7.3.

First, let us focus on the execution time. As expected, given $dMax$, increasing k systematically leads to better performance. Moreover, increasing the budget tends to decrease the variance between measurements, i.e., estimates become more reliable. However, despite multiplying the number of random walks by ten, the gain in terms of execution time is not that large, especially on Virtuoso. When using a bottom-up approach (as a DP algorithm), the quality of join orders mainly depends on the quality of the first joins [83]. Thus, accuracy on 1-way, 2-way, or 3-way joins is often enough to get good join orders and does not require a large budget.

If increasing the budget always leads to better performance, increasing $dMax$ may negatively impact the execution time. Random walks are uniformly distributed over the interval $1..dMax$. The larger the interval is, the fewer walks remain to estimate each part, i.e., the more inaccurate the estimator is. Consequently, with a small budget, it is better to reduce $dMax$ to have more accurate cardinalities. Even if property path patterns may be underestimated (because they are not fully explored), estimates will be more reliable. However, with a larger budget, it is worth looking for a larger $dMax$ to better capture the real cost of property path patterns. The budget k being the most impacting factor in optimization time, a good strategy to define k and $dMax$ is to define a budget first, and then select $dMax$ by testing different values until the quality of join orders deteriorates. For instance, with a budget of 1000 random walks on Blazegraph, setting $dMax = 5$ results in good performance, but increasing $dMax$ to 10 starts deteriorating the execution time.

7.6 Conclusion

In this chapter, we have introduced a new query optimizer that relies on the relaxation of reachability-patterns and a sampling-based cardinality estimator to find efficient join orders for $C2RPQ_{UF}$ queries. The experimental study demonstrates that our proposal outperforms existing engines. On the newly proposed Wikidata Query Benchmark, the workload execution time is divided by 14 compared to Blazegraph. This work opens several perspectives. First, optimization times can be significantly improved using a better budget model [83]. Moreover, computing the confidence interval of estimators [85] may allow us to adapt the budget to each join order rather than systematically computing k random walks. Generalizing our approach to nested stars is also part of our research agenda. Finally, our proposal relies on random walks to estimate the cardinality of $C2RPQ_{UF}$ queries. Another exciting line of research would be to study how random walks can be used to estimate the cardinality of SPARQL queries in the presence of the MINUS, OPTIONAL, and FILTER NOT EXISTS operators.

Part III

Improving the Scalability of Federation Engines

Chapter 8

Introduction

In this part, we focus on the scalability issue of SPARQL federation engines. Existing federation engines, such as FedX [118], CostFed [112], and SemaGrow [38], have been mostly evaluated on small federations, i.e., less than 20 knowledge graphs. As a result, very little is known about the ability of federation engines to scale when the number of sources increases.

To illustrate, we reuse the example of the FedShop benchmark [42] provided in the introduction of this thesis. FedShop is a federated benchmark that simulates an e-commerce scenario, with online shops and rating sites. Each shop and site is a knowledge graph that is connected to others through similar products. FedShop queries simulate users searching for products, reviews, or similar products across the federation. For example, Query $Q5$ in Figure 8.1a supposes that a user is looking for similar products to those she is interested in. We executed Query $Q5$ over two federations of 20 (10 shops and 10 rating sites) and 200 (100 shops and 100 rating sites) sources, and compared the performance of CostFed [112], i.e., the best-performing federation engine to date, with hand-crafted SERVICE queries. The SERVICE query for $Q5$ is provided by the Reference Source Assignment (RSA) in FedShop. Table 8.1b reports CostFed and RSA execution times for $Q5$ on both federations.

1. In terms of performance, the RSA outperforms CostFed on both federations. In the case of the 20-source federation, the RSA is roughly 60 times faster than CostFed, whereas across the larger 200-source federation, it exhibits a minimum speed advantage of 3600 times. Waiting more than 1 hour to get similar products is unrealistic in a dynamic scenario like an e-commerce application.
2. Regarding scalability, when we increase the federation size by a factor of 10, the RSA execution time increases by a factor of 30. In contrast, the execution time of CostFed multiplies by a minimum of 120.

The question is, why such a performance gap?

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?product ?localProductLabel WHERE {
    ?localProductXYZ owl:sameAs bsbm:Product43923   ;
        bsbm:productPropertyNumeric1 ?origProperty1 ;
        bsbm:productPropertyNumeric2 ?origProperty2 ;
        bsbm:productFeature ?localProdFeatureXYZ .

    ?localProdFeatureXYZ owl:sameAs ?prodFeature .
    ?localProdFeature owl:sameAs ?prodFeature .

    ?localProduct bsbm:productFeature ?localProdFeature ;
        bsbm:productPropertyNumeric1 ?simProperty1 ;
        bsbm:productPropertyNumeric2 ?simProperty2 ;
        rdfs:label ?localProductLabel ;
        owl:sameAs ?product .

    FILTER(bsbm:Product43923 != ?product)
    FILTER(?simProperty1 < (?origProperty1 + 20) && ?simProperty1 > (?origProperty1 - 20))
    FILTER(?simProperty2 < (?origProperty2 + 70) && ?simProperty2 > (?origProperty2 - 70))
}
```

(a) FedShop Query $Q5$. Returns similar products to Product43923.

	20 shops	200 shops
RSA	50ms	1.5 seconds
CostFed	3 seconds	$>$ 1 hour

(b) Execution time of Query $Q5$ on 20 and 200 shops using CostFed and hand-crafted (RSA) query execution plans.

Figure 8.1 – Execution time of FedShop Query $Q5$ on 20 and 200 shops using CostFed and hand-crafted (RSA) query execution plans.

Federation engines performance are closely tied to the problem of finding optimal source assignments [39]. Given a set of SPARQL endpoints and a SPARQL query, a source assignment maps each part of the query to SPARQL endpoints on which they must be executed to get complete results. An optimal source assignment minimizes the number of SPARQL endpoints to contact, i.e., the number of sub-queries sent to the federation. Minimizing the number of sub-queries means pushing as much computation as possible to the local endpoints. In other words, sending queries to where the data is. In contrast, a poor source assignment may lead a federation engine to transfer all the intermediate results of a SPARQL query, drastically impacting performance.

Unfortunately, finding optimal source assignments is NP-hard [39]. Consequently, federation engines rely on best-effort strategies that may produce suboptimal solutions.

In this part, we propose FedUP, a novel approach that finds better source assignments than existing engines. The key idea is to start from the solutions mappings of the query. By identifying the sources involved in the production of each solution mappings individually, it is possible to build source assignments beyond the reach of current SPARQL federation engines. Of course, computing the solutions mappings of the query is not realistic. To tackle this issue, FedUP relies on random walks to sample SPARQL queries. In doing so, FedUP may generate incomplete results, if limited to a fixed budget, but will eventually return complete results as the budget increases.

The remaining chapters of this part are organized as follows. First, in chapter 9, we review existing federation engines. Then, in chapter 10 we present our proposal FedUP.

Chapter 9

Querying Decentralized Knowledge Graphs Online: State of the art

Contents

Numerous approaches have been proposed for executing SPARQL queries across a federation of SPARQL endpoints [3, 38, 45, 52, 93, 103, 110, 112, 118]. Most federation engines start with an exhaustive source assignment, e.g., all federation members are relevant for all triple patterns, which ensures complete and correct results. Then, the initial source assignment is refined through pruning and grouping steps in order to minimize the number of sub-queries that will be sent to the federation.

9.1 Triple-Pattern-Wise Source Selection

To illustrate, suppose that we start with a simple source assignment where all federation members are considered relevant for all triple patterns. Such a simple source assignment produces a correct joins-over-unions query plan [39], and returns complete and correct results. However, the source assignment may be suboptimal.

A first simple pruning step is to consider that all federation members may not be relevant for all triple patterns. A triple pattern should only be evaluated on SPARQL endpoints that would return non-empty results. Such a triple-pattern-wise source selection is achieved by sending ASK queries to the federation members [118], or using data summaries [52].

A first simple grouping step is to group together triple patterns that share the same

single relevant source. Such groups of triple patterns are called exclusive groups [118]. Triple patterns that belong to the same exclusive group can be evaluated together using a single sub-query.

9.2 Join-Aware Triple-Pattern-Wise Source Selection

Pruning can be further improved using join-aware source selection algorithms [110, 112]. Given two triple patterns along with their respective sources, the goal is to prune sources that do not contribute to the join results. HiBiScuS [110] and CostFed [112] relies on data summaries to compute BGP-aware source selections.

Grouping can be further improved when a global join variable can be safely evaluated as a local join variable, i.e., the join variable can be evaluated on remote endpoints without impacting the completeness of query results. Lusail [1] is able to determine if a global join variable can be evaluated as a local join variable by sending simple SPARQL queries to the federation. Such a grouping technique can significantly reduce the number of sub-queries sent to endpoints.

Whatever the approach, it is not clear how close from the minimal source assignment the pruning/grouping paradigm can converge. To prune a source assignment, traditional approaches must ensure that it does not contribute to the final results. FedUP takes the opposite direction. FedUP adds a new source only when it is *certain* that this source contributes to the final results. FedUP starts from the query results obtained through random walks to incrementally build source assignments. With a limited budget, random walks may miss relevant endpoints and produce incomplete results. However, FedUP eventually converges to a logical query plan that produces complete and correct results as the budget increases.

ANAPSID [3, 94] also proposes a trade-off between completeness and performance, but unlike FedUP, it uses heuristics that do not allow converging toward completeness.

Regarding the dichotomy between zero-knowledge approaches such as FedX and Lusail, and summary-based approaches such as SemaGrow [38] and CostFed; FedUP works without prior knowledge about the data stored at endpoints. Thus, FedUP works both in a zero-knowledge setting, and with data summaries to speed up source selection but at the expanse of accuracy. Specifically, random walks fit any summary that is a graph homomorphism of the federated graph [37].

Chapter 10

FedUP: A Federated SPARQL Query Engine Powered by Random Walks

10.1 Background and Motivations

In this chapter, we use the concepts of RDF and core SPARQL [100, 61], i.e., triple patterns (tp), basic graph patterns (BGP), `AND`, `UNION`, `FILTER`, and `OPTIONAL` as defined in chapter 2.

We assume RDF graphs to be accessible through SPARQL endpoints (or sources). Let $\mathcal{D}$ be the set of endpoints, a federation F is defined as a finite subset of $\mathcal{D}$. For each endpoint $D_i \in \mathcal{D}$, we write G_i to denote the underlying RDF graph exposed by D_i. The evaluation $[\![P]\!]_F$ of a SPARQL graph pattern P over a federation F results in a set of solution mappings identical to the evaluation of P over the union of graphs composing the federation: $[\![P]\!]_F = [\![P]\!]_{G_F}$, where $G_F = \bigcup_{D_i \in F} G_i$. To execute a query over a federation, the query first has to be decomposed into sub-queries that are associated with endpoints selected for executing the sub-queries[2]. This decomposition can be expressed with the language of FedQPL [39, 40].

FedQPL [39, 40] defines source assignments through expressions φ representing core SPARQL operators. The set of its terminal symbols is the following: (i) Symbol $req_{D_Y}^{tpX}$, also noted $X@Y$, requests the execution of triple pattern tpX at federation member D_Y. (ii) Symbol *mj* Φ and (iii) Symbol *mu* Φ are multi-join and multi-union respectively. They accept any number of expressions Φ and do not enforce any execution order. (iv) Symbol $\mathit{leftjoin}(\varphi_1, \varphi_2)$ and (v) Symbol $\mathit{filter}^R(\varphi)$ represents their respective SPARQL counterparts `OPTIONAL` and `FILTER`.

To illustrate, let us consider the federation F described in Figure 10.1 and the query S_6 of Figure 10.1. F includes 4 independent SPARQL endpoints D_1, D_2, D_3, and D_4.

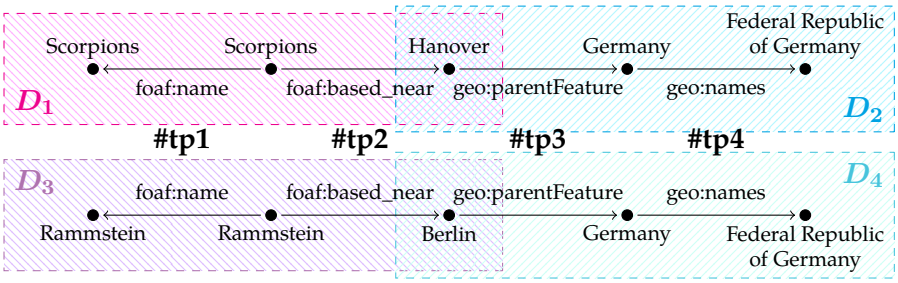


Figure 10.1 – Example of a federation comprising 4 RDF graphs linked by common vertices. Vertex $Hanover$ is duplicated and links D_1 and D_2; and Vertex $Berlin$ is duplicated and links D_3 and D_4, inspired by FedBench [116].

The entity `Scorpions` of D_1 is referencing the entity `Hanover` hosted in D_2. The entity `Rammstein` of D_3 references the entity `Berlin` of D_4. Evaluating S_6 on F returns 2 solution mappings : $\mu_1 = \{?artist \rightarrow$ `Scorpions`, $?name \rightarrow$ `Scorpions`, $?location \rightarrow$ `Hanover`, $?germany \rightarrow$ `Germany`$\}$ and $\mu_2 = \{?artist \rightarrow$ `Rammstein`, $?name \rightarrow$ `Rammstein`, $?location \rightarrow$ `Berlin`, $?germany \rightarrow$ `Germany`$\}$.

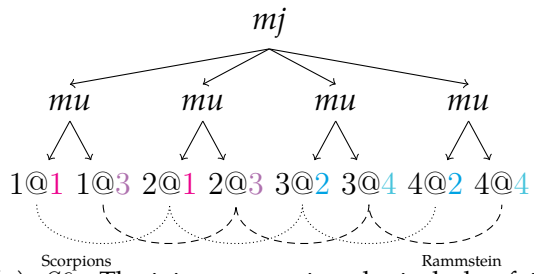
Federation engines source assignments [112, 118] generate joins-over-unions logical plans $S_{\bowtie_{(\cup)}}$ that are of the form $mj\ \{(mu\ \{(req_D^{tp})^+\})^+\}$ [39, 40]. Figure 10.2a depicts a joins-over-unions logical plan for Query $S6$ that would be generated by the state-of-the-art federation engine CostFed [112]. The FedQPL expression for describing this plan is:

$$S6_j = mj\ \{mu\ \{req_{D_1}^{tp1}, req_{D_3}^{tp1}\}, mu\ \{req_{D_1}^{tp2}, req_{D_3}^{tp2}\}, mu\ \{req_{D_3}^{tp3}, req_{D_2}^{tp3}\}, mu\ \{req_{D_2}^{tp4}, req_{D_4}^{tp4}\}\}$$

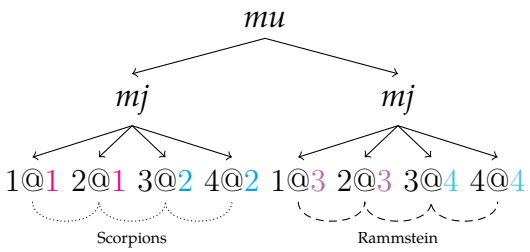
A common improvement of such an initial decomposition is to merge sub-queries that are associated with the same single data source; such merged sub-queries are called exclusive groups [118]. As all triples are associated to 2 endpoints, exclusive groups cannot be found. Figure 10.2b illustrates an alternative plan that generates the same results as the plan depicted in Figure 10.2a but belongs to the class of unions-over-joins logical plans $S_{\cup_{(\bowtie)}}$ that are of the form $mu\ \{(mj\ \{(req_D^{tp})^+\})^+\}$. The FedQPL expression for describing this plan is:

$$S6_u = mu\ \{mj\ \{req_{D_1}^{tp1}, req_{D_1}^{tp2}, req_{D_2}^{tp3}, req_{D_2}^{tp4}\}, mj\ \{req_{D_3}^{tp1}, req_{D_3}^{tp2}, req_{D_4}^{tp3}, req_{D_4}^{tp4}\}\}$$

Exclusive groups are easily marked, which allows us to rewrite the previous FedQPL



(a) $S6_j$: The joins-over-unions logical plan fails to capture the relationship between bindings. The query execution must check 16 combinations while only 2 are relevant.



(b) $S6_u$: The unions-over-joins logical plan provides 2 sub-trees accurately capturing the 2 combinations required to create the results.

Figure 10.2 – Logical plans for query S6 over the F federation expressed in FedQPL Language.

expression as:

$$S6_u = mu\ \{mj\ \{req_{D_1}^{tp1,tp2}, req_{D_2}^{tp3,tp4}\}, mj\ \{req_{D_3}^{tp1,tp2}, req_{D_4}^{tp3,tp4}\}\}$$

Query $S6_u$ is better than $S6_j$ since it generates only 4 sub-queries instead of 8. Both presented source assignments are minimal in their class, but some minimal plans are inaccessible to joins-over-unions [39]. From the plan $S6_u$, it is possible to generate the joins-over-unions plan $S6_j$, but the converse is false: generating a union of conjunctions starting from $S6_j$ will generate all 16 combinations of conjunctions. Currently, no engine can produce efficient unions-over-joins plans as the one of $S6_u$.

Problem *How to produce efficient unions-over-joins logical plans ?*

10.2 FedUP: A Unions-over-Joins Federation Engine

FedUP stands for Federated Unions-over-joins Plans query engine. FedUP relies on unions-over-joins logical plans that allow for the independent execution of sub-queries while favoring the identification of exclusive groups. FedUP performs random walks over federated graphs or summaries to retrieve samples of the query results. As many query results share the same source assignment, FedUP is able on the current benchmark to quickly gather enough solution mappings to converge to correct and complete source assignments.

Algorithm 10: Source assignment $\mathcal{A}$ for a query Q over a federation F.

1 **Function** $\mathcal{A}(Q, F)$**: return** *mu* $\mathcal{A}'(Q, F)$ ▷ Root of the logical plan
2 **Function** $\mathcal{A}'(P, F)$**:** ▷ Explores every graph pattern P
3 **if** P is a triple pattern tp **then**
4 **return** $\{req_D^{tp} \mid D \in F \wedge sols(req_D^{tp}) \neq \varnothing\}$
5 **else if** P is (P_1 `AND` P_2) **then** ▷ $P_1 \bowtie P_2$
6 $\Phi_1, \Phi_2 \leftarrow \mathcal{A}'(P_1, F), \mathcal{A}'(P_2, F)$
7 **return** $\{mj\ \{\varphi_1, \varphi_2\} \mid \varphi_1 \in \Phi_1, \varphi_2 \in \Phi_2 \wedge sols(mj\ \{\varphi_1, \varphi_2\}) \neq \varnothing\}$
8 **else if** P is (P_1 `UNION` P_2) **then** ▷ $P_1 \cup P_2$
9 $\Phi_1, \Phi_2 \leftarrow \mathcal{A}'(P_1, F), \mathcal{A}'(P_2, F)$
10 **return** $\Phi_1 \cup \Phi_2$
11 **else if** P is (P_1 `OPTIONAL` P_2) **then** ▷ $P_1 ⟕ P_2$
12 $\Phi_1, \Phi_2, \Phi_{opt} \leftarrow \mathcal{A}'(P_1, F), \mathcal{A}'(P_2, F), \varnothing$
13 **for** $\varphi_1 \in \Phi_1$ **do**
14 $\Phi_{join} \leftarrow \{\varphi_2 \mid \varphi_2 \in \Phi_2 \wedge sols(mj\ \{\varphi_1, \varphi_2\}) \neq \varnothing\}$
15 **if** $\Phi_{join} = \varnothing$ **then** $\Phi_{opt} \leftarrow \Phi_{opt} \cup \{\varphi_1\}$
16 **else** $\Phi_{opt} = \Phi_{opt} \cup \{leftjoin(\varphi_1, mu\ \Phi_{join})\}$
17 **return** Φ_{opt}
18 **else if** P is (P' `FILTER` R) **then**
19 $\Phi = \mathcal{A}'(P', F)$
20 **return** $\{filter^R(\varphi) \mid \varphi \in \Phi \wedge sols(filter^R(\varphi)) \neq \varnothing\}$

10.2.1 Building Unions-over-Joins Logical Plans from Results

To build a unions-over-joins source assignment for a query Q, a simple but unrealistic idea is to evaluate the query Q on an exhaustive source assignment, track the provenance of results, and build a unions-over-joins source assignment. Starting from the results ensures that only conjunctions that return results are included in the union of joins. Even though it is unrealistic, such an approach allows us to find a correct and complete unions-over-joins source assignment for Q.

Algorithm 10 builds a source assignment for query Q over a federation F based on a set of recursive rules. Some rules use $sols(\varphi)$ as a function that returns the mappings, resulting in the evaluation of the expression φ over F.

We illustrate this algorithm on Query $S6$ of Figure 10.3 over Federation F_{ex} of Figure 10.1. First the algorithm merges all upcoming subexpressions with a multi-union at Line 1. Then, it enters Line 5 with ($tp1$ `AND` ($tp2$ `AND` ($tp3$ `AND` $tp4$))). Line 2 states that evaluating $tp3$ and $tp4$ both returns $\{req_{D_2}^{tp}, req_{D_4}^{tp}\}$. Then, Line 7 checks that their

```
SELECT * WHERE {
    ?artist foaf:name ?name . #tp1
    ?artist foaf:based_near ?location . #tp2
    ?location geo:parentFeature ?germany . #tp3
    ?germany geo:name "Federal Republic of Germany" . #tp4
}
```

Figure 10.3 – Query S6 from FedBench [116] that requests the name of artists located near the Federal Republic of Germany.

intersections actually return mappings. Here, *mj* $\{req_{D_2}^{tp3}, req_{D_2}^{tp4}\}$ and *mj* $\{req_{D_4}^{tp3}, req_{D_4}^{tp4}\}$ indeed return mappings, but most importantly *mj* $\{req_{D_2}^{tp3}, req_{D_4}^{tp4}\}$ and *mj* $\{req_{D_4}^{tp3}, req_{D_2}^{tp4}\}$ do not. Only the former expressions are kept, the latter ones are discarded. After applying the joining rule for every triple pattern and simplifying nested multi-join expressions, we obtain the expected plan of Figure 10.2b with 4 exclusive groups, i.e., `tp1.tp2` at D_1 and D_3, `tp3.tp4` at D_2 and D_4:

$$mu\ \{mj\ \{req_{D_1}^{tp1}, req_{D_1}^{tp2}, req_{D_2}^{tp3}, req_{D_2}^{tp4}\}, mj\ \{req_{D_3}^{tp1}, req_{D_3}^{tp2}, req_{D_4}^{tp3}, req_{D_4}^{tp4}\}\}$$

While SPARQL operators `AND`, `UNION`, and `FILTER` produce unions-over-joins logical query plans easily, `OPTIONAL` introduces additional complexity due to the lack of distributivity and partial mappings. Generated plans with `OPTIONAL` are unions-over-joins-based plans with a slightly different grammar to include *leftjoin*:

$$S_{\mathcal{A}} \begin{cases} a_u ::= & a_j \mid mu\ \Phi_u \\ a_j ::= & a_b \mid mj\ \Phi_b \mid leftjoin(a_j, a_u) \\ a_b ::= & req_D^{tp} \end{cases}$$

To illustrate, we modify the federation in Figure 10.1. We assume that D_1, D_3, and D_4 are merged into a graph D_{134} thus containing *Rammstein*'s full mappings plus *Scorpions*' partial mappings. We define Query $S6'$ as : `SELECT * WHERE {tp1.tp2 OPTIONAL {tp3.tp4}}`. Figure 10.4 depicts the logical plan built by Algorithm 10. ($tp1$ `AND` $tp2$) and ($tp3$ `AND` $tp4$) are evaluated to $\{mj\ \{req_{D_{134}}^{tp1}, req_{D_{134}}^{tp2}\}\}$, and $\{mj\ \{req_{D_{134}}^{tp3}, req_{D_{134}}^{tp4}\}$, $mj\ \{req_{D_2}^{tp3}, req_{D_2}^{tp4}\}\}$ respectively. Federation engines must take into account that the optional part of the query (`tp3.tp4`) requires 2 federated members (D_2 and D_{134}). They cannot evaluate it as independent sub-queries at D_2 and D_{134}. Otherwise, they would

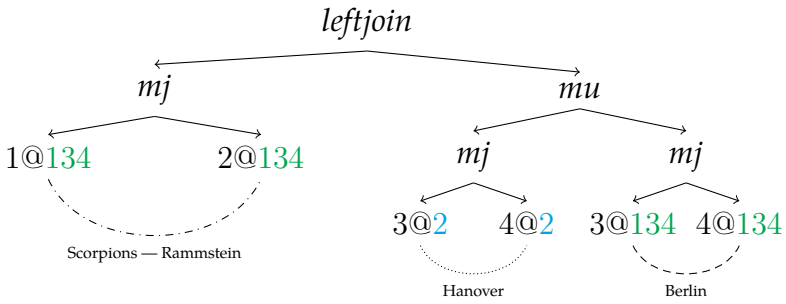


Figure 10.4 – Logical plan of $S6'$: `SELECT * WHERE {tp1.tp2 OPTIONAL {tp3.tp4}}`.

produce incorrect results such as: {*?artist* → `Scorpions`, *?name* → `Scorpions`, *?location* → `Hanover`}. This partial mappings should not exist since it has links to `Germany` in the federation. To avoid this issue, Line 18 evaluates the `OPTIONAL` operator. For each left expression P_1, multi-union merges the right expressions P_2 that return mappings. Then, a terminal symbol *leftjoin* links each left expression with its corresponding right expressions. In this example, 2 out of 4 combinations return results. Figure 10.4 shows that Algorithm 10 creates an unions-of-joins with 2 multi-joins. Federation engines can efficiently execute this logical plan since it helps to identify 3 exclusive groups: `tp1.tp2` at D_{134}, `tp3.tp4` at D_2 and D_{134}.

10.2.2 Building Unions-over-Joins Logical Plans with Random Walks

Building plans as proposed in Algorithm 10 requires knowledge of solution mappings, as highlighted in Lines 4,7,14,20, which is not affordable. Nevertheless, assuming that many solution mappings share the same combination of sources, sampling a small fraction of the query results is sufficient to build its source assignment.

FedUP exploits random walks to sample query results in a pay-as-you-go manner. Algorithm 11 implements budgeted random walks guided by a query Q over a graph F representing the federation. Function $\mathcal{A}-$RWs creates a sample graph using k random walks, then calls $\mathcal{A}$ (Algorithm 10) to create a logical plan. The function RW is a recursive function that explores every graph pattern of a query. It returns a solution mappings and a label λ that maps each triple pattern of the query to the corresponding source that produced the mappings. At its core, Line 6 handles triple patterns by selecting a random source from the set of *relevant* sources, then a random triple from this source and pattern. The function *relevant* can be implemented in various ways, e.g., by performing `ASK` queries to remote sources or by querying summaries of these remote

graphs. A random walk may fail to provide a correct mappings. Nevertheless, over several walks, our approach iteratively improves its approximation of source assignment and eventually builds an unions-over-joins logical plan that provides a correct and complete result.

Algorithm 11: Approximating the source assignment of a query Q over a federation F with a budget of k random walks.

Function $\mathcal{A}$-RWs(Q, F, k)**:**
  $\Omega \leftarrow \bigcup_{i=1}^{k} \text{RW}(Q, F)$ ▷ Random walking with a budget
  $F' \leftarrow \{\mathcal{T}(G, \Omega) \mid G \in F\}$ ▷ Create a federation using the walks
  return $\mathcal{A}(Q, F')$ ▷ Build source assignement with limited knowledge
Function RW(P, F)**:** ▷ Random walk in every graph pattern P
  if P is a triple pattern tp **then**
    $D \leftarrow random(relevant(tp, F))$ ▷ $\approx R_Q(tp)$ for F
    $\mu \leftarrow random(sols(req_D^{tp}))$
    return $\{(\mu, \lambda[tp \rightarrow D])\}$
  else if P is (P_1 `AND` P_2) **then** ▷ $P_1 \bowtie P_2$
    $\Omega_1 \leftarrow \text{RW}(P_1, F)$
    if $\Omega_1 \neq \varnothing$ **then**
      $\Omega_2 \leftarrow \text{RW}(\mu_1(P_2), F)$ where $(\mu_1, \lambda_1) \in \Omega_1$
      return $\{(\mu_1 \cup \mu_2, \lambda_1 \cup \lambda_2) \mid (\mu_1, \lambda_1) \in \Omega_1 \wedge (\mu_2, \lambda_2) \in \Omega_2\}$
    return Ω_1
  else if P is (P_1 `UNION` P_2) **then** ▷ $P_1 \cup P_2$
    return $\{(\mu, \lambda) \in random(\text{RW}(P_1, F) \cup \text{RW}(P_2, F))\}$
  else if P is (P_1 `OPTIONAL` P_2) **then** ▷ $P_1 ⟕ P_2$
    $\Omega_1 \leftarrow \text{RW}(P_1, F)$
    if $\Omega_1 \neq \varnothing$ **then**
      $\Omega_2 \leftarrow \text{RW}(\mu_1(P_2, F))$ where $(\mu_1, \lambda_1) \in \Omega_1$
      if $\Omega_2 \neq \varnothing$ **then**
        return $\{(\mu_1 \cup \mu_2, \lambda_1 \cup \lambda_2) \mid (\mu_1, \lambda_1) \in \Omega_1 \wedge (\mu_2, \lambda_2) \in \Omega_2\}$
    return Ω_1
  else if P is (P' `FILTER` R) **then**
    return $\{(\mu, \lambda) \mid (\mu, \lambda) \in \text{RW}(P', F) \wedge R(\mu)\}$
Function $\mathcal{T}(G, \Omega)$**:** ▷ Keep mappings that belong to a graph G
  return $\{\mu(tp) \mid (\mu, \lambda) \in \Omega, tp \in Q, vars(tp) \subseteq dom(\mu) \wedge \lambda(tp) = G\}$

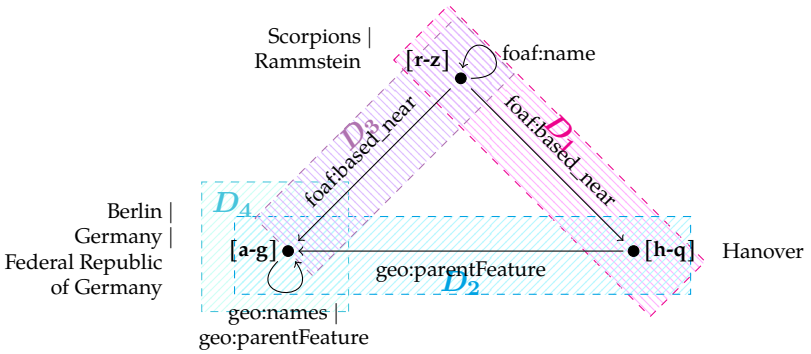


Figure 10.5 – Quotient summary of Figure 10.1's graph. Vertices are merged together depending on the first letter of their label or value.

10.2.3 Random Walks on Summaries

Algorithm 11 requires a graph F as a union of graphs representing the federation. However, copying all the graphs of the federation is unrealistic for large federation [1]. Fortunately, the versatility of random walks allows them also to work (1) online where F becomes an interface to remote graphs or (2) on quotient summaries where F is a homomorphic graph of the union of graphs, much smaller but also much denser [1]: there are fewer vertices that have more edges between each other.

Figure 10.5 depicts an example of a quotient summary of Figure 10.1's graph. It groups `Berlin`, `Germany`, and `Federal Republic of Germany` into Vertex `[a-g]`; `Hanover` into Vertex `[h-q]`; `Scorpions` and `Rammstein` into Vertex `[r-z]`. Even in this toy example, the resulting graph is denser than its original version. Querying this graph also requires changes in the query. In Figure 10.3, Query $S6$ remains identical, except for the literal `Federal Republic of Germany` that becomes `[a-g]`. To construct an unions-over-joins source assignment, one could execute the modified query over the summary. However, dense graphs may lead to combinatorial explosions that preclude such a naive approach. On the other hand, dense graphs have many combinations that lead to identical source assignments. Random walks over summaries can quickly explore them all.

Using Algorithm 11 on Figure 10.5's summary graph and its modified Query $S6$, every random walk succeeds in providing meaningful insights for source assignment. For instance, to perform a random walk, it first stops at Line 6 to process `tp1: ?artist foaf:name ?name`. Relevant sources for this triple pattern are D_1 and D_3. Assuming it selects D_3, it then selects a random triple from this source: `[r-z] foaf:name [r-z]`. Second, Algorithm 11 joins it at Line 10 with `tp2`. Again, 2 sources are candidates but target

different objects. Assuming it selects D_1, the chosen triple is `[r-z] foaf:based_near [h-q]`. Third, it joins the result with `tp3`: D_2's triple `[h-q] geo:parentFeature [a-g]`. Finally, it joins the result with `tp4`, which implies choosing between D_2, and D_4. Assuming it selects D_2, the result of this random walk implicitly states that $sols(mj\ \{req_{D_3}^{tp1}, req_{D_1}^{tp2}, req_{D_2}^{tp3}, req_{D_2}^{tp4}\}) \neq \varnothing$, which is accurate on the summary but incorrect on the federated graph. Overall, Algorithm 10 creates a multi-union of 12 multi-joins while only 2 are necessary on the federated graph. It remains better than the joins-over-unions plan that implies 16 multi-joins.

Random walks allow federation engines to build logical plans based on federated graphs or summaries. The next section aims to evaluate the performance of random walks on source selection, and the performance of our source selection on federated query execution.

10.3 Experimental Study

FedUP operates by building unions-over-joins logical plans using random walks. This experimental study aims at answering the following questions: (1) Does FedUP perform better than existing engines overall? (2) How much does FedUP's plans contribute to its query execution time? (3) How much time do random walks need to guarantee complete and correct results?

10.3.1 Experimental Setup

We compared FedUP to 3 state-of-the-art federation engines, namely (i) the historical representative FedX [118], (ii) a summary-based representative HiBiScuS [110], (iii) and CostFed [112], the current best-performing federation engine (up to our knowledge). To test different trade-offs, we evaluated 2 configurations of FedUP: (iv) FedUP-id for which random walks are computed over the most accurate summary, i.e., a local dump of the federation graphs (v) FedUP-auth for which random walks are computed over a quotient summary of the federation graphs, built in the same spirit as HiBiScuS's summary, i.e., entities with the same URL authorities are projected on the same vertex, while all literals are projected to a single vertex.

HiBiScuS and CostFed have been implemented on top of FedX [1]. To keep the same

1. `https://github.com/dice-group/CostFed`

execution environment, we decided to use FedX as our query engine as well. FedX handles the optimization of queries, the computation of exclusive groups, the decomposition into sub-queries, and the evaluation of sub-queries on the relevant endpoints. In our experiments, we stored all federation graphs on different named graphs of a single Virtuoso endpoint (Version 7.2.7.3234-pthreads). To compute random walks over summaries, we enhanced Apache Jena (Version 4.7.0) to provide random triples on their default backend called TDB. The random walks computed by Jena are then used to build the unions-over-joins logical plans evaluated by FedX.

State-of-the-art approaches ensure complete and correct results. To compare FedUP with such federation engines, we configured FedUP to run until it finds a source assignment that provides complete and correct results. Because FedUP does not use `ASK` queries to refine its source assignments, we configured HiBiScuS and CostFed to use the index-dominant version of their source selection algorithm, i.e., in our experiments, HiBiScuS and CostFed do not use any `ASK` query. Regarding FedX, we configured it to clear its cache after each query, which is the worst-case scenario for its source selection.

To run our experiments, we used a local cloud instance with Ubuntu 20.04.4 LTS, a AMD EPYC 7513-Core processor with 16 vCPUs allocated to the VM, 1TB SSD, and 64GB of RAM. The Virtuoso endpoint hosting the data, as well as the federation engines, ran on the same machine. Code, configurations, queries, and datasets can be found on the GitHub platform at `https://github.com/GDD-Nantes/FedUP-experiments`.

10.3.2 LargeRDFBench: No Engine Stands Out

To demonstrate that FedUP does not degrade performance on existing benchmarks, we first ran experiments on LargeRDFBench. LargeRDFBench is the most commonly used benchmark to evaluate the performance of federation engines [1, 38, 110, 112, 118]. The benchmark is explicitly designed to represent federated SPARQL queries on real-world datasets. In our experiments, we used 24 queries over 14 endpoints that include all types of core SPARQL operators, such as `UNION`, `OPTIONAL` and `FILTER`.

Figure 10.6 presents the performance achieved by the different approaches. For each query, the x-axis displays competitors, while the y-axis displays the time spent performing source selection, and executing queries, in seconds on a logarithmic scale. Each query ran 5 times, and measurements were averaged before being reported in Figure 10.6. Each query is executed with a timeout of 20 minutes.
Focusing on source selection, Figure 10.6 shows that, surprisingly, both FedUP-id and

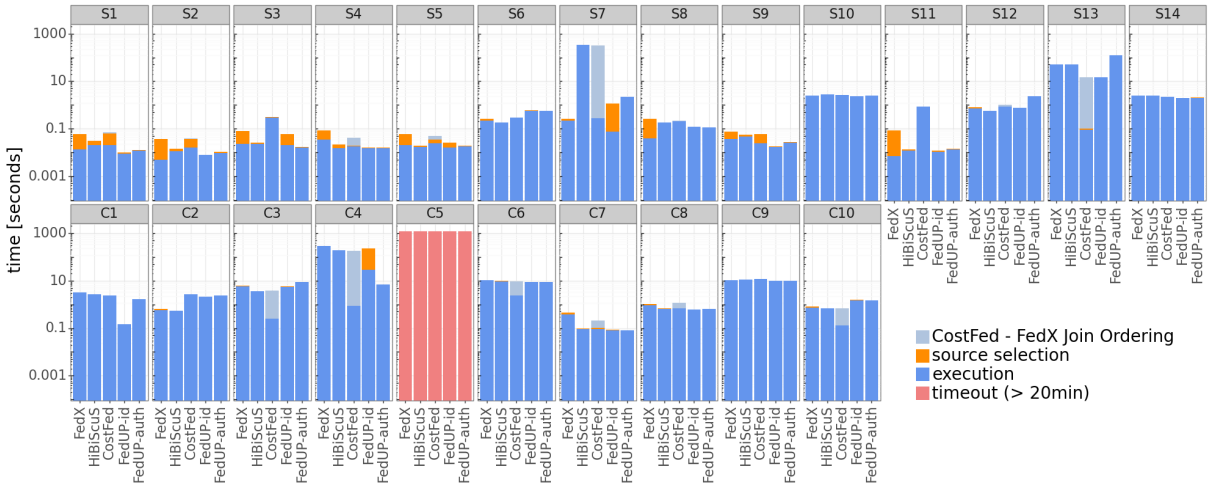


Figure 10.6 – Average performance achieved by FedX, HiBiScuS, CostFed, and FedUP on LargeRDFBench queries.

FedUP-auth are very efficient in finding complete and correct source assignments. Looking for complete source assignments is a worst-case scenario for random walks. If each source assignment corresponds to a unique solution mappings, random walks end up looking for all solution mappings, which may be significantly more expansive than evaluating the query. Nevertheless, LargeRDFBench queries return much more solution mappings than there are source assignments. For instance, Query $S10$ has a unique source assignment but returns 9054 solutions mappings, and finding only one of them is enough to get a complete source assignment. Query $C4$ is slightly more challenging for FedUP-id since it needs to find 64 source assignments. On the FedUP-auth's summary, source selection queries are less selective than on FedUP-id, making the search easier. However, random walks on summaries may find source assignments that return no results when actually executed on the federation. Despite returning no result, evaluating them may be costly. There exists a trade-off between an accurate summary on which random walks may have a hard time finding assignments but that returns solutions, and a small summary on which random walks are very efficient but can generate many false positive assignments.

Focusing on execution time, Figure 10.6 shows that, except for CostFed that sometimes outperforms its competitors because of a better join ordering [112], all approaches perform similarly when evaluated using the same join ordering. Indeed, 20 of the 24 queries have only one relevant source per triple pattern. In this context, joins-over-unions and unions-over-joins logical plans are identical.

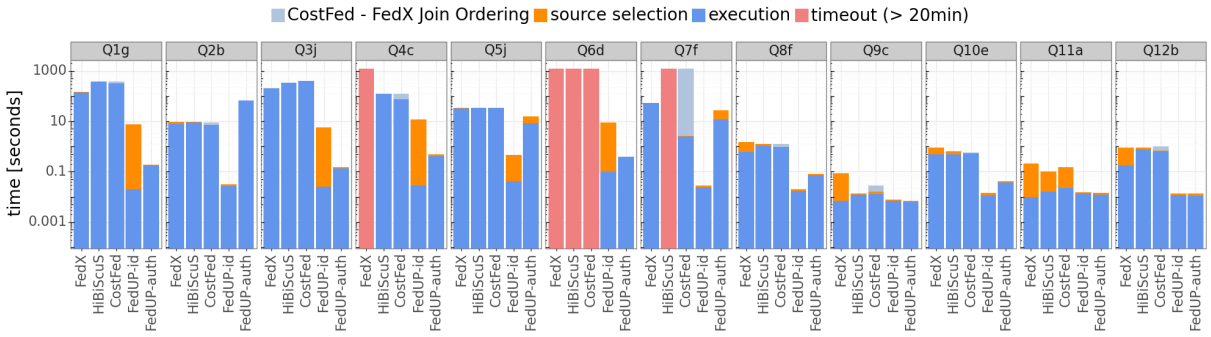


Figure 10.7 – Average performance achieved by FedX, HiBiScuS, CostFed, and FedUP on FedShop queries for the 100 endpoints use-case.

10.3.3 FedShop: FedUP Finds Better Logical Plans

LargeRDFBench is not challenging enough to differentiate an approach relying on joins-over-unions logical plans from another relying on unions-over-joins plans. To highlight the differences between approaches, we ran experiments on FedShop [42], a benchmark designed to study the scalability of federation engines. We used the 12 queries and the 100 endpoints use-case presented in the recent hackathon on querying federations of knowledge graphs [2]. While LargeRDFBench focuses on queries that return a single source assignment involving multiple sources, FedShop focuses on queries that return multiple source assignments but involving a few endpoints.

Figure 10.7 presents the performance achieved by the different approaches on FedShop. As expected, FedUP performs significantly better than state-of-the-art approaches. On FedShop, state-of-the-art approaches end up with joins-over-unions logical plans where multiple endpoints are assigned to each triple pattern, preventing FedX from computing exclusive groups. All joins are performed locally, drastically degrading performance. For instance, FedX, HiBiScuS, and CostFed all timeout on $Q6d$, which comprises only 2 triple patterns. They failed to identify that both triple patterns must be executed together on remote endpoints. Even with source assignments that are not minimal, finding exclusive groups makes the difference. For all queries but $Q6d$, FedUP-auth and HiBiScuS finds the same relevant endpoints for triple patterns, but only FedUP-auth can find exclusive groups.

Figure 10.7 shows significantly higher source selection times for FedUP-id on FedShop than on LargeRDFBench. For queries $Q1g$, $Q3j$, $Q4c$, and $Q6d$, FedUP-id even takes

2. `https://github.com/MaastrichtU-IDS/federatedQueryKG`

more time looking for source assignments than executing queries. These queries all include selective filters that challenge random walks. Random walks are efficient on triple patterns because they rely on SPO, POS, and OSP indexes to pick random triples from the triples that match triple patterns. However, indexes cannot be used to pick random triples that match some filter expression. Consequently, selective filters can drastically increase the number of failed random walks, negatively impacting source selection times.
Overall, FedUP provides better total execution times. Most often, the smaller query execution times come from better logical plans that make up for the time taken to build these plans. Without obvious optimizations (e.g., join order, biased random walks, parallelism), FedUP already exposes promising results for federated query processing, and highlights the need for plans that further exploit exclusive groups.

10.4 Conclusion

In this chapter, we have presented FedUP, our proposal to the problem of source selection and query decomposition in federation engines. Instead of pruning coarse source assignments, FedUP refines its plans using random walks in a pay-as-you-go fashion. The flexibility of random walks allows FedUP to work on the federated graph or their quotient summaries. Assuming that many query results share identical combinations of sources, our approach quickly builds unions-over-joins plans that retrieve most of the results. These plans have better parallelization capabilities and facilitate the identification of exclusive groups, further leveraging the processing power of remote endpoints. With an unlimited budget, FedUP guarantees logical plans that retrieve correct and complete results. Evaluation results on federated benchmarks confirmed the effectiveness of the proposed approach. Compared to state-of-the-art engines [112, 118], FedUP finds better source assignments with more exclusive groups, leading to better query execution times.

We plan to investigate strategies for optimizing random walk exploration to improve our approach further. Successful exploration relies on both the ability to find helpful information and the ability to explore different paths. The join order and cardinality of triple patterns heavily influence this. We aim to introduce a dynamic bias to improve the chances of discovering rare source combinations and achieving more efficient exploration.

Part IV

Conclusion and Perspectives

CONCLUSION AND PERSPECTIVES

10.5 Conclusion

We identified two research questions for querying online decentralized knowledge graphs:

1. How to provide a fair SPARQL query service that ensures complete results?
2. How to improve the scalability of SPARQL federation engines?

Providing a fair SPARQL query service that returns complete results. There is a tension between 3 key properties of public SPARQL endpoints; the fairness of the server, the completeness of query results, and the execution time of queries. We conjecture that one property has to be sacrificed to ensure the others. For example, the fair use policies enforced by public SPARQL endpoints sacrifice the completeness of query results to be fair and deliver good performance. However, sacrificing completeness prevents many crucial use cases and limits the adoption of the Semantic Web. LDF interfaces sacrifice performance to be fair and ensure complete results. If performance are not overly impacted, sacrificing performance can be an acceptable trade-off. Consequently, the major contribution of this thesis is to minimize the loss of performance while guaranteeing fairness and completeness.

Web Preemption is the approach that best minimizes the performance overhead [92] compared with LDF interfaces. However, Web Preemption does not effectively support COUNT-DISTINCT and property path queries. To tackle this issue, we first extended Web Preemption to improve execution performance of COUNT-DISTINCT queries:

> Julien Aimonier-Davat et al., « Online approximative SPARQL query processing for COUNT-DISTINCT queries with web preemption », *in*: *Semantic Web* 13.4 (2022), pp. 735–755, DOI: 10.3233/SW-222842

Computing an exact COUNT-DISTINCT requires transferring a large amount of data. The key idea is to approximate COUNT-DISTINCT queries, but with bounded error rates to keep the results meaningful. This is possible using probabilistic data structures [50]. Knowing the number of distinct values in a dataset is crucial for many

use cases. For instance, it can help the query planner optimize SPARQL queries for better performance [51, 112]. Additionally, such statistics are used to compute VoID descriptions [70, 89], and to perform basic statistical analyses. For example, COUNT-DISTINCT queries can be used to determine whether there are an equal number of men and women working as professors in French universities. With this contribution, we drastically reduce COUNT-DISTINCT queries data transfer and clients memory consumption, enabling the computation of such queries on online knowledge graphs.

Next, we extended Web Preemption to improve execution performance of SPARQL property path queries:

> Julien Aimonier-Davat, Hala Skaf-Molli, and Pascal Molli, « Processing SPARQL Property Path Queries Online with Web Preemption », *in*: *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021*, vol. 12731, 2021, pp. 57–72.

The key idea is to support Partial Transitive Closures on the server, and let the Web client decompose transitive closure expressions into many Partial Transitive Closures. Supporting Partial Transitive Closures on the server drastically improves query execution performance [8]. Being able to efficiently execute property path queries on online knowledge graphs is of paramount importance. Indeed: (1) property path queries are widely used. For instance, they represent 38% of the Wikidata query logs [34]; (2) property paths enable users to navigate complex taxonomy, hierarchies, and relationships defined in ontologies; (3) property paths are essential for tasks like pathfinding and graph traversal in Semantic Web applications. For example, we can use property path queries to find paths between resources or to discover related resources through a series of intermediate predicates. Thanks to this contribution, it is now possible to compute property path queries online, and get complete results with performance close to current SPARQL endpoints.

During this thesis, we analysed the performance of many queries from many benchmarks. As pointed out by Leis et al. [84], many improvements made on operators are counter-balanced by bad join orders. To address this issue, we proposed a new query optimizer to find better join orders for SPARQL queries, especially in the presence of property paths:

> Julien Aimonier-Davat et al., « Join Ordering of SPARQL Property Path Queries », *in*: *The Semantic Web - 20th International Conference, ESWC 2023, Hersonissos, Crete, Greece, May 28 - June 1, 2023, Proceedings*, vol. 13870, 2023, pp. 38–54.

The key idea is to relax property path patterns to make their actual cost observable

by existing cost models. This contribution allows query optimizers to correctly order property path patterns in basic graph patterns. With small changes on servers, we significantly improve execution time for property path queries. For example, we reduce Blazegraph query execution time by a factor of at least 14 [10].

Improving the scalability of SPARQL federation engines. The FedShop [42] benchmark has highlighted the lack of scalability of existing SPARQL federation engines. Currently, federation engines are limited to small federations, i.e., less than 20 endpoints, while the LOD Cloud has a thousand endpoints. To address this limitation, we introduced FedUP, a new federation engine. The key idea is to generate union-over-joins query execution plans [39], rather than generating join-over-unions plans as other engines do. FedUP drastically improves federated query performance by finding source assignments beyond the reach of existing federation engines. Thanks to FedUP, we can now target federations of the size of the Linked Open Data Cloud.

10.6 Perspectives

Short-term perspectives. Thanks to the contributions of this thesis, the Web Preemption approach should now be able to support all the WDBench [17] queries efficiently, i.e., the Wikidata workload. However, it requires integrating all our contributions into a reliable resource that the Semantic Web community can use. To achieve this objective, we decided to extend the Apache Jena server with Web Preemption and a sampling interface.

- The sampling interface is already released in:

  > Julien Aimonier-Davat et al., « RAW-JENA: Approximate Query Processing for SPARQL Endpoints », *in*: *The Semantic Web - ISWC 2023 - 22th International Semantic Web Conference, Athens, Greece, November 6-10, 2023, Proceedings*, 2023

  This extension allows Apache Jena to support sampling approximate query processing (S-AQP). In this thesis, we use S-AQP to optimize join orders and compute FedUP source assignments. However, S-AQP enables many other use cases such as computing large-scale statistics [124], knowledge graph embeddings [107], approximate aggregations [140], summaries [72] , and exploratory queries [5].

- The Web Preemption model has already been implemented in Apache Jena and currently supports core SPARQL. Implementing Web Preemption only requires overriding Jena operators with equivalent preemptable operators.

The short-term objectives are to provide a fair SPARQL query service built on top of Apache Jena and to perform extensive experimental studies to compare Web Preemption with LDF interfaces and SPARQL endpoints. We expect Web Preemption to deliver high performance in all situations, whether facing high query loads or operating without any load.

Long-term perspectives. Regarding fair query processing, the Web Preemption model can still be improved, e.g., by adding new operators on the server or improving existing operators. During this thesis, we extended Web Preemption with a partial TOP-k operator and early-pruning:

> Julien Aimonier-Davat, Hala Skaf-Molli, and Pascal Molli, « Processing SPARQL TOP-k Queries Online with Web Preemption », *in*: *Proceedings of the QuWeDa 2022: 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs co-located with 21st International Semantic Web Conference (ISWC 2022), Hangzhou, China, 23-27 October 2022*, vol. 3279, 2022, pp. 33–48.

While early-pruning significantly improves execution performance of TOP-k queries, supporting early-termination could allow us to achieve even better performance. A first interesting perspective is to study how to support early-termination of TOP-k queries in the context of Web Preemption. Combining preemptable operators also raise interesting questions. For example, ranking solutions mappings based on the result of aggregate functions may raise an issue as aggregate functions are only partially executed on the server. Similar questions arise for TOP-k and aggregate queries containing property paths.

Regarding the join ordering of SPARQL queries, the current approach neither supports property paths with nested transitive closures nor MINUS, OPTIONAL and EXISTS operators [10]. Supporting nested transitive closures could be done using rewriting, but a simpler and more elegant solution would be to represent property paths as finite-state automata. Random walks could then be adapted to perform walks on the product of a graph and an automaton. The budget model can also be improved to significantly reduce optimization times. For example, we could fix a budget after which we would rely on basic statistics and a greedy algorithm to guarantee bounded optimization times [83]. Moreover, not all join orders need the same number of random walks

to get accurate estimates. An interesting perspective could be to propose an adaptive budget model. For example, using confidence intervals [85] to decide which join order to sample and how many random walks to perform. Confidence intervals could also be used to distribute random walks between sub-queries. Finally, another interesting future work is estimating the cardinality of a SPARQL query containing MINUS, OPTIONAL, and EXISTS operators. Such operators raise new issues because of their semantics. For example, the OPTIONAL operator may lead a random walk to succeed when it should not have, misleading estimators.

FedUP opens up a wide range of possibilities for future work. A first perspective could be to improve the way we compute random walks. For example, by pondering the probability of drawing a triple for a specific source with the number of triples on that source. The aim would be to balance the probabilities between (over and under)-represented sources to find source assignments faster. A second perspective could be to target federated query processing with zero-knowledge, e.g., by computing random walks on remote endpoints that support S-AQP. While a zero-knowledge solution could be costly, it does not require acquiring and maintaining summaries. Another perspective could be to focus on query optimization. While FedUP outperforms existing federation engines, union-over-joins query execution plans may be suboptimal. It may exist better query execution plans that are neither union-over-joins nor join-over-unions. However, finding those plans is still an open question.

Part V

Résumé en langue française

Chapter 11

Introduction

11.1 Graphes de connaissances

Le Web sémantique est né avec Tim-Berners Lee [29], la personne qui a inventé le World Wide Web dans les années 1990. L'objectif principal du Web sémantique est de rendre le Web accessible aux ordinateurs, facilitant ainsi le partage de connaissances entre institutions, entreprises et individus. Le Web peut alors être perçu comme une immense "base de données" à laquelle n'importe quel ordinateur peut poser des questions afin d'accomplir des tâches sophistiquées pour les utilisateurs [29].

Pour concrétiser cette vision, la communauté du Web sémantique a mis en place différentes normes, telles que l'utilisation du framework RDF (Resource Description Framework) pour représenter les connaissances [91], et celle du langage de requête SPARQL pour les interroger [61].

Grâce à RDF, les graphes de connaissances (Knowledge Graphs, KGs) peuvent être définis comme une collection de faits. Par exemple, le graphe de connaissances DBpedia [82] est un ensemble de faits extrait depuis Wikipédia et représenté au format RDF. La Figure 1.1 correspond à un fragment de DBpedia qui fournit des informations sur Barack Obama, l'ancien président des États-Unis. Dans cette représentation, les informations sur le président Obama sont structurées sous forme de triples (Sujet, Propriété, Valeur), chaque triple affirmant un fait spécifique. Le premier triple indique que Barack Obama est une personne. Le deuxième nous révèle son nom de naissance, tandis que les triples suivants nous informent qu'il a été président des États-Unis après George W. Bush, et qu'il a écrit des livres tels que "Dream from My Father" et "A Promised Land".

À l'heure actuelle, le graphe de connaissances DBpedia contient près de 9,5 milliards de triples RDF, ce qui représente une portion importante des connaissances de Wikipédia dans différentes langues. Dans un graphe de connaissances, chaque ressource est identifiée par une URI. Par exemple, la ressource correspondant à l'entité Barack Obama sur DBpedia est associée à l'URI `https://dbpedia.org/resource/Barack_Obama`. En

```
@prefix dbo: <http://dbpedia.org/ontology/>                          .
@prefix dbr: <http://dbpedia.org/resource/>                          .
@prefix dbp: <http://dbpedia.org/property/>                          .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>           .
@prefix owl: <http://http://www.w3.org/2002/07/owl#>                 .

dbr:Barack_Obama    rdf:type    dbo:Person                           ;
    dbp:birthName   "Barack Hussein Obama II"                        ;
    dbp:title       dbr:President_of_the_United_States               ;
    dbp:before      dbr:George_W._Bush                               ;
    dbo:author      dbr:Dreams_from_My_Father, dbr:A_Promised_Land    .

dbr:Barack_Obama    owl:sameAs  <http://viaf.org/viaf/52010985>       ,
    <http://d-nb.info/gnd/132522136>                                 ,
    <http://data.europa.eu/euodp/jrc-names/Barack_Obama>             ,
    <http://musicbrainz.org/artist/...7abdc144f1d5>                  ,
    <http://www.wikidata.org/entity/Q76>                             ,
    <http://data.bibliotheken.nl/id/thes/p287770850>                 .
```

Figure 11.1 – Représentation textuelle de triples RDF extrait depuis DBpedia et décrivant l'entité Barack Obama.

déréférençant cette URI, nous accédons à l'ensemble des triples que possède DBpedia sur l'ancien président des États-Unis.

Une solution simple pour interroger un graphe de connaissances consiste à utiliser le langage de requête SPARQL. Par exemple, si nous souhaitons savoir qui est devenu président des États-Unis après George W. Bush, nous pouvons écrire la requête SPARQL présentée dans la Figure 11.2. Cette requête implique la jointure du résultat de deux triple patterns. Le premier triple pattern recherche les présidents des États-Unis, tandis que le second recherche les successeurs de George W. Bush. Dans le cadre de cette requête, la variable $?s$ est l'entité que nous recherchons, c'est-à-dire une entité définie comme étant à la fois un président et un successeur de George W. Bush. Pour exécuter cette requête, nous pouvons utiliser le SPARQL endpoint public de DBpedia, accessible à l'adresse `https://dbpedia.org/sparql#`. Un SPARQL endpoint est un serveur Web conçu pour exécuter des requêtes SPARQL sur un graphe de connaissances. Le principal avantage des SPARQL endpoints est de fournir une interface standardisée [48] pour interroger des données RDF. En exécutant notre requête sur DBpedia, nous obtenons

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

SELECT ?s WHERE {
    ?s  dbp:title   dbr:President_of_the_United_States  .
    ?s  dbp:before  dbr:George_W._Bush                  .
}
```

Figure 11.2 – Une requête SPARQL qui récupère le successeur de George W. Bush à la présidence des États-Unis.

le résultat suivant: `http://dbpedia.org/resource/Barack_Obama`.

Grâce aux recommandations du W3C, de nombreux graphes de connaissances sont aujourd'hui accessible via des SPARQL endpoints. Parmis les graphes de connaissances les plus populaires auxquels on peut accéder en ligne, on retrouve DBpedia et Wikidata. Dbpedia et Wikidata contiennent tous deux des données encyclopédiques, et permettent de répondre à diverses requêtes. Ces requêtes peuvent aller de l'énumération des descendants de Bach, à la découverte des rues en France portant des noms de femmes célèbres, jusqu'à la liste des ponts qui enjambent les rivières de la ville de Leipzig en Allemagne [1].

11.2 Graphes de connaissances décentralisés

Le véritable intérêt des graphes de connaissances réside dans leur capacité à se connecter les uns aux autres. Comme on peut le voir dans la Figure 11.1, les derniers triples utilisent la propriété *sameAs* pour lier la ressource `http://dbpedia.org/entity/Barack_Obama` à d'autres ressources décrivant la même entité, c'est-à-dire Barack Obama, dans d'autres graphes de connaissances. Chacune de ces ressources nous permet d'en apprendre davantage sur Barack Obama. Par exemple, le graphe de connaissances du New York Times nous fournit la liste des articles de presse concernant l'ancien président des États-Unis, tandis que le graphe de connaisances de MusicBranz nous donne des informations sur la discographie de Barack Obama. En connectant ses ressources à

1. `https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`

d'autres ressources, DBpedia embrasse les principes du Linked Data proposés par Tim Berners-Lee [28]: (1) Utiliser des URI pour identifier les entités; (2) Utiliser le protocole HTTP afin que les personnes puissent rechercher ces entités; (3) Lorsque quelqu'un recherche une URI, fournir des informations en suivant les normes (e.g., RDF); (4) Inclure des liens vers d'autres URI afin de permettre la découverte de plus d'informations.

Les principes du Linked Data ont pour objectif d'aider la communauté à publier, partager, et interconnecter des données sur le Web. À l'heure actuelle, des centaines de graphes de connaissances ont déjà été publié en suivant ces principes [30, 114]. Tous ces graphes de connaissances interconnectés les uns aux autres créent un vaste graphe de connaissances décentralisé avec lequel tout le monde peut interagir, et auquel tout le monde peut contribuer. Ce graphe est communément appelé le Linked Open Data Cloud (LOD Cloud). L'état actuel du LOD Cloud est représenté dans la Figure 11.3. À la date du 1er septembre 2023, il contient 1314 ensembles de données, 16308 liens qui connectent ces différents ensembles, et couvre de nombreux domaines de connaissances. Par exemple, on peut trouver des graphes de connaissances qui contiennent des données institutionnelles, des données linguistiques, des données sur les sciences de la vie, ou des connaissances plus générales telles que des connaissances encyclopédiques.

En utilisant le langage de requête SPARQL, n'importe quel utilisateur peut interroger un ou plusieurs SPARQL endpoints. Par exemple, la requête illustrée dans la Figure 11.4a interroge à la fois DBpedia et Wikidata pour extraire des informations sur Barack Obama. Pour spécifier sur quel SPARQL endpoint telle ou telle partie de la requête doit s'exécuter, nous pouvons utiliser les clauses SERVICE introduites dans le langage à partir de SPARQL 1.1 [61]. Dans la requête de la Figure 11.4a, la première clause SERVICE est exécutée sur Wikidata, afin de récupérer toutes les informations correspondant à la ressource `https://www.wikidata.org/entity/Q76`, c'est-à-dire Barack Obama sur Wikidata. La deuxième clause est exécutée sur DBpedia, et utilise le lien *sameAs* de la Figure 11.1 pour trouver la ressource `http://dbpedia.org/entity/Barack_Obama` sur DBpedia, et extraire les informations de DBpedia sur Barack Obama. Nous appelons les requêtes comme celle décrite dans la Figure 1.4a des *requêtes SERVICE*.

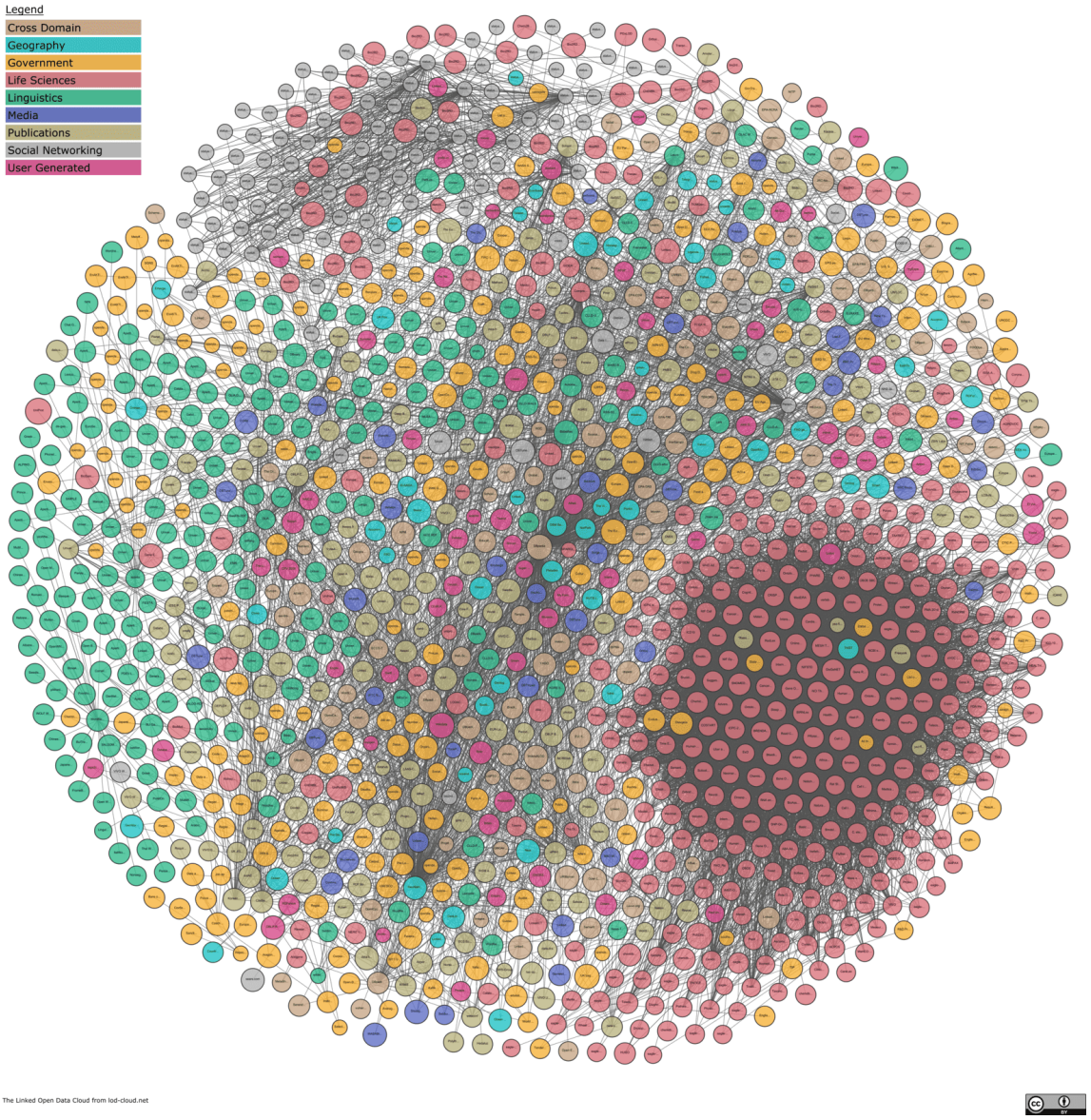


Figure 11.3 – Représentation graphique du LOD Cloud tel qu'il est à la date du 1er septembre 2023.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX wd: <http://www.wikidata.org/entity/>

SELECT ?p ?o WHERE {
  {
    SERVICE <https://query.wikidata.org> {
      wd:Q76 ?p ?o .
    }
  } UNION {
    SERVICE <https://dbpedia.org/sparql> {
      ?s owl:sameAs wd:Q76 . ?s ?p ?o .
    }
  }
}
```

(a) Service query

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX wd: <http://www.wikidata.org/entity/>

SELECT ?p ?o WHERE {
  {
    wd:Q76 ?p ?o .
  } UNION {
    ?s owl:sameAs wd:Q76 . ?s ?p ?o .
  }
}
```

(b) Federated query

Figure 11.4 – Requête SPARQL qui retourne toutes les informations que détiennent Wikidata and DBpedia sur Barack Obama.

11.3 Moteurs de requête fédérée

Interroger plusieurs graphes de connaissances à l'aide de requêtes SERVICE peut rapidement devenir fastidieux lorsque le nombre de graphes de connaissances augmente. Par exemple, pour trouver toutes les informations concernant Barack Obama sur le LOD Cloud, nous devons (1) chercher l'ensemble des graphes de connaissances qui sont pertinent pour notre requête, (2) contacter les différents SPARQL endpoints, c'est-à-dire étendre la requête présentée dans la Figure 11.4a avec une clause SERVICE pour chacun des SPARQL endpoints à contacter.

Réaliser un tel traitement à la main est irréaliste. Heureusement, ce traitement peut être automatisé en utilisant un moteur de requête fédérée. Un moteur de requête fédérée prend en entrée une requête SPARQL classique, ainsi qu'une liste de SPARQL endpoints, et retourne une requête SERVICE, avec la garantie que tous les SPARQL endpoints qui contribuent à la requête seront interrogés. Par exemple, en fournissant à un moteur de requête fédérée la requête présentée dans la Figure 11.4b, ainsi que les SPARQL endpoints DBpedia et Wikidata, ce dernier est en mesure de retrouver la requête SERVICE présentée dans la Figure 11.4a. Dans le contexte des moteurs de requête fédérée, on appel la liste des SPARQL endpoints une *fédération*. Ainsi, utiliser un moteur de requête fédérée permet à n'importe quel utilisateur d'interroger une fédération de manière transparente.

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

SELECT ?creativeWork ?fictionalWork WHERE {
    ?creativeWork wdt:P31/wdt:P279* wd:Q17537576 . # creative work
    ?creativeWork wdt:P144 ?fictionalWork .
    ?fictionalWork wdt:P136 wd:Q8253 . # fiction
}
```

Figure 11.5 – Requête $Q1$. Récupère la liste des oeuvres créatives sur Wikidata qui se sont inspirées d'un oeuvre de fiction.

Les moteurs de requête fédérée jouent un rôle crucial dans le monde du Web sémantique. En permettant à n'importe quel utilisateur d'interroger l'ensemble des SPARQL endpoints publics disponibles, ils contribuent à préserver la nature décentralisé du Web sémantique.

11.4 Les problèmes des moteurs de requête fédérée

Malheureusement, évaluer une requête SPARQL sur une fédération de SPARQL endpoints publics soulève de nombreuses problématiques: (1) Les SPARQL endpoints publics ne garantissent pas des résultats complets (2) Les moteurs de requête fédérée ne passent pas à l'échelle quand le nombre de sources augmente.

Les SPARQL endpoints ne garantissent pas des résultats complets Les moteurs de requête fédérée font l'hypothèse que les SPARQL endpoints retournent des résultats complets, ce qui n'est pas vrai en pratique [92]. Un service de requête SPARQL public doit faire face à des charges imprévisibles de requêtes SPARQL. Le principal défi pour un service public est alors de garantir que le service reste disponible malgré des variations importantes dans les taux d'arrivée des requêtes et les ressources nécessaires au traitement des requêtes. Les fournisseurs de données font souvent référence à ce défi comme "la mise en place d'une politique d'utilisation équitable". Pour atteindre une politique d'utilisation équitable, la plupart des SPARQL endpoints publics imposent des quotas. Par exemple, le SPARQL endpoint public de Wikidata interrompt toutes les requêtes qui sont encore en cours d'exécution après 60 secondes. Les quotas garantissent que les ressources du serveur sont partagées équitablement entre les utilisateurs.

En contrepartie, ils peuvent empêcher une requête de retourner des résultats complets. Par exemple, la requête $Q1$ présentée dans la Figure 11.5 retourne la liste des oeuvres créatives sur Wikidata qui se sont inspirées d'une oeuvre de fiction. Si nous tentons d'exécuter $Q1$ sur le SPARQL endpoint public de Wikidata, la requête est interrompue après 60 secondes, et ne retourne aucun résultat. Malheureusement, $Q1$ n'est pas un cas isolé; de nombreuses requêtes sur Wikidata rencontrent des problèmes similaires [33]. Selon les logs de Wikidata [2], des milliers de requêtes sont interrompues chaque mois parce qu'elles dépassent la limite en temps imposée par le SPARQL endpoint. Sans surprise, parmis les requêtes les plus souvent interrompues, on retrouve les requêtes property path et les requêtes d'agrégation. Le problème est tel que le dernier benchmark sur Wikidata met principalement l'accent sur les requêtes property path [17]. Les graphes de connaissances ne saissant de croître, et les requêtes de se complexifier [14, 101], nous nous attendons à ce que de plus en plus de requêtes soient interrompues par les politiques d'utilisation équitable des SPARQL endpoints publics.

Ne pas être en mesure de garantir des résultats complets est une limitation stricte de la part des SPARQL endpoints. Sans cette garantie, il est impossible pour une application de se reposer sur cette technologie. Pour illustrer, certaines applications utilisent la description VoID [13] des graphes de connaissances pour acquérir des informations sur leur contenu. Si elle n'est pas fournie, la description VoID peut être calculée en exécutant des requêtes SPARQL [70]. Malheureusement, la plupart de ces requêtes sont interrompues par la politique de partage équitable des SPARQL endpoints [70, 89]. Le même problème touche les moteurs de requête fédérée qui ont besoin d'acquérir un résumé des données pour fonctionner. C'est le cas par exemple d'HiBiscuS [110] et de CostFed [112].

Indéniablement, les SPARQL endpoints existants sont précieux. Cependant, dans leur quête pour fournir un service équitable, ils ont dû faire un compromis, et ont décidé de sacrifier la complétude des résultats. Parce que garantir des résultats complets est crucial pour de nombreux cas d'utilisation, le premier défi de cette thèse est de fournir *un service de requête SPARQL équitable qui garantit des résultats complets*.

Les moteurs de requête féderée ne passent pas à l'échelle En supposant que les SPARQL endpoints publics parviennent à fournir des résultats complets, le défi pour un moteur de requête fédérée est alors d'être capable de passer à l'echelle quand le

2. `https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en`

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?product ?localProductLabel WHERE {
    ?localProductXYZ owl:sameAs bsbm:Product43923   ;
        bsbm:productPropertyNumeric1 ?origProperty1 ;
        bsbm:productPropertyNumeric2 ?origProperty2 ;
        bsbm:productFeature ?localProdFeatureXYZ .

    ?localProdFeatureXYZ owl:sameAs ?prodFeature .
    ?localProdFeature owl:sameAs ?prodFeature .

    ?localProduct bsbm:productFeature ?localProdFeature ;
        bsbm:productPropertyNumeric1 ?simProperty1 ;
        bsbm:productPropertyNumeric2 ?simProperty2 ;
        rdfs:label ?localProductLabel ;
        owl:sameAs ?product .

    FILTER(bsbm:Product43923 != ?product)
    FILTER(?simProperty1 < (?origProperty1 + 20) && ?simProperty1 > (?origProperty1 - 20))
    FILTER(?simProperty2 < (?origProperty2 + 70) && ?simProperty2 > (?origProperty2 - 70))
}
```

(a) Requête $Q5$ du benchmark FedShop. Retourne la liste des produits similaires au produit Product43923.

	20 shops	200 shops
RSA	50ms	1.5 seconds
CostFed	3 seconds	$>$ 1 hour

(b) Temps d'exécution de la requête $Q5$ sur une fédération de 20 et 200 sources.

Figure 11.6 – Temps d'exécution de la requête $Q5$ du benchmark FedShop, sur une fédération de 20 et 200 sources, avec d'un côté CostFed et de l'autre le RSA.

nombre de sources augmente [1, 42].

Pour illustrer ce point, intéressons-nous au benchmark FedShop [42]. FedShop est un benchmark fédérée qui simule un scénario de commerce électronique; avec des boutiques en ligne et des sites d'évaluation. Chaque boutique ou site est un graphe de connaissances connecté à d'autres par le biais de produits similaires. Les requêtes FedShop simulent des utilisateurs à la recherche d'un produit, d'une liste de produits similaires, ou d'avis. Par exemple, la requête $Q5$ présentée dans la Figure 11.6a suppose qu'un utilisateur est à la recherche de produits ayant des caractéristiques similaires à un produit de référence. Nous avons exécuté la requête $Q5$ sur deux fédérations; une de 20 sources (10 boutiques et 10 sites d'évaluation) et une de 200 sources (100 boutiques et 100 sites d'évaluation). Sur ces deux fédérations, nous avons comparé les performances

de CostFed [112], c'est-à-dire le moteur de requête fédérée le plus performant à ce jour, avec une requête SERVICE construite à la main, qui apparaît sous le nom de RSA dans la Figure 11.6. La Table 11.6b dans la Figure 11.6 nous donne le temps d'exécution pour CostFed et le RSA sur les deux fédérations. À partir de ces résultats, on peut tirer les conclusions suivantes:

1. Premièrement, CostFed souffre d'un important problème de performance. Sur une fédération de 20 sources, CostFed est 60 fois plus lent que le RSA, et cet écart passe de 60 à plus de 3600 sur une fédération de 200 sources. Pour un utilisateur d'une application de commmerce électronique, de telles performances sont inaceptables.
2. Deuxièmement, CostFed ne passe pas à l'échelle quand le nombre de sources augmente. Lorsque la taille de la fédération est multiplié par 10, le temps d'exécution de CostFed est multiplié par 120 au minimum, soit 4 fois plus que le RSA au minimum.

Tandis que le LOD Cloud compte des centaines de graphes de connaissances, les résultats obtenus avec FedShop montrent clairement que les moteurs de requête fédérée actuels sont limités à la gestion de petites fédérations. Heureusement, les performances du RSA prouvent qu'il est possible de grandement améliorer les moteurs de requête fédérée. Par conséquent, le deuxième défi de cette thèse est de *résoudre les problèmes de performance et de passage à l'échelle des moteurs de requête fédérée*.

11.5 Questions de recherche

Dans cette thèse, nous cherchons à répondre aux questions de recherche suivantes:

1. Comment fournir un service de requête SPARQL qui garantit des résultats complets tout en étant équitable?
2. Comment améliorer le passage à l'échelle des moteurs de requête fédérée?

11.5.1 Comment fournir un service de requête SPARQL qui garantit des résultats complets tout en étant équitable?

Les SPARQL endpoints publics ont fait un choix délibéré: sacrifier la complétude des résultats pour garantir un partage équitable des ressources du serveur. Cependant,

en faisant ce sacrifice, ils limitent drastiquement les cas d'utilisation et l'adoption du Web sémantique.

Plutôt que de sacrifier la complétude des résultats, un autre compromis est de sacrifier un peu de performances pour garantir à la fois l'équité du serveur et des résultats complets. Différentes approches ont été proposé, offrant différents compromis entre équité et performances [4, 23, 24, 67, 134]. Le principe sur lequel repose ces approches est de répartir la charge que représente le traitement des requêtes entre le serveur et le client. Ainsi, le serveur peut se restreindre à un sous-ensemble des opérateurs du langage SPARQL pour lesquels il peut garantir un partage équitable de ses ressources, tandis que le client prend en charge les opérateurs restants. Cependant, distribuer la charge entre le client et le serveur introduit un overhead important sur le réseau, et donc sur les performances de ces approches. Pour minimiser cet overhead, la solution consiste à implémenter le plus d'opérateurs possible sur le serveur, tout en préservant l'équité du serveur afin de garantir des résultats complets.

Parmis les solutions proposées, la préemption Web est celle qui minimise le mieux l'overhead réseau [92]. Contrairement aux autres approches, la préemption Web propose un nouveau modèle d'exécution pour les moteurs de requête SPARQL. Ce modèle permet à un moteur de requête, i.e., et par extension un service de requête, de suspendre une requête en cours d'exécution, avec pour objectif de reprendre son exécution plus tard. Cette mécanique de Stop-and-Go permet au serveur de supporter une grande partie des opérateurs du langage SPARQL, tout en restant équitable afin de garantir des résultats complets. Toutefois, cette approche à une limite. Pour être viable, le serveur doit minimiser le temps qu'il passe à suspendre et à reprendre l'exécution des requêtes. Ansi, les opérateurs qui ne permettent pas une reprise efficace ne sont pas supportées par le serveur. De ce fait, les requêtes property path et COUNT-DISTINCT doivent être partiellement évaluées sur le client, dégradant les performances du moteur pour ces requêtes. Améliorer les performances des requêtes property path et COUNT-DISTINCT est primordial, car elles jouent un rôle central dans de nombreux cas d'utilisation. Pour pallier à ce problème, nous étendons le modèle de la préemption Web. L'objectif est d'ajouter un support côté serveur pour déléguer une plus grande partie de l'évaluation sur le serveur.

Outre la préemption Web, l'ordre dans lequel les différentes clauses sont exécutées dans une requête, communément appelé l'*ordre de jointure*, a un impact significatif sur le temps d'exécution des requêtes [84]. Sans un bon ordre de jointure, une requête qui

s'exécute en quelques secondes peut mettre des jours à s'exécuter. Bien que la préemption Web permet de garantir des résultats complets peu importe le temps d'exécution de la requête, il est primordial de trouver les bons ordres de jointure afin d'offrir un service efficace. Dans cet objectif, nous proposons un nouvel algorithme capable de trouver de bons ordres de jointure pour les requêtes SPARQL conjonctives avec des filtres et des expressions property path.

11.5.2 Comment améliorer le passage à l'échelle des moteurs de requête fédérée?

Les performances des moteurs de requête fédérée sont étroitement liées au problème de la sélection des sources et de la décomposition des requêtes [39]. Étant donné une requête SPARQL et un ensemble de SPARQL endpoints, le problème est le suivant: quelles parties de la requête doivent être évaluées sur quels SPARQL endpoints pour obtenir des résultats complets, tout en minimisant le nombre de sous-requêtes? Minimiser le nombre de sous-requêtes revient à déléguer autant de calcul que possible sur les SPARQL endpoints. Autrement dit, cela permet d'amener les requêtes aux données et non l'inverse.

Pour améliorer le passage à l'échelle des moteurs de requête fédérée, nous proposons FedUP. FedUP est un moteur de requête fédérée qui utilise des marches aléatoires afin d'identifier comment les sources qui sont pertinentes pour une requête sont combinées pour produire les solutions de la requête. Grâce à cette information, FedUP est capable de trouver une meilleure décomposition que les moteurs de requête fédérée existant.

11.6 Mes publications

(1) **RAW-JENA : Un support pour l'échantillonnage de requête sur les SPARQL endpoints publics**, par Julien Aimonier-Davat, Minh-Hoang Dang, Pascal Molli, Brice Nédelec et Hala Skaf-Molli. Dans International Semantic Web Conference, 2023.
Résumé : Avoir un support pour l'échantillonnage de requête sur un SPARQL endpoint public ouvrirait la porte à de nombreuses applications, telles que calculer des statistiques, des résumés, ou des embeddings sur des graphes de connaissances en ligne. Un tel support permettrait également de chercher de meilleurs ordres de join-

ture, calculer des agrégations en ligne, ou encore explorer les graphes de connaissances. Cependant, malgré sa compatibilité avec les politiques de partage équitable des SPARQL endpoints, aucun serveur n'offre un support pour l'échantillonnage de requête SPARQL. Dans cette démonstration, on présente RAW-JENA: une extension d'Apache Jena qui offre un support en ligne pour l'échantillonnage de requête SPARQL via l'utilisation de marches aléatoires. L'objectif de RAW-JENA est de mettre en avant l'intérêt et la faisabilité d'offrir un tel support pour un moteur de requête SPARQL.

(2) **FedShop : Un benchmark pour tester la scalabilité des moteurs de requête fédérée**, par Minh-Hoang Dang, Julien Aimonier-Davat, Pascal Molli, Olaf Hartig, Hala Skaf-Molli et Yotlan Le Crom. Dans International Semantic Web Conference, 2023.
Résumé : Bien que plusieurs approches ont été proposé pour interroger une fédération de SPARQL endpoints, très peu d'information sont disponibles sur le comportement des moteurs de requête fédérée quand la taille de la fédération augmente. En effet, les benchmarks existant ne permettent pas de faire varier la taille des fédérations. Pour combler ce vide, nous proposons un nouveau benchmark, spécialement conçu pour tester le comportement d'un moteur de requête fédérée sur des fédérations de différentes tailles. Sur la base de ce benchmark, nous avons étudié le comportement de quatre moteurs de requête fédérée. Les résultats de nos expériences montre clairement que les moteurs de requête fédérée existants ne sont pas capable de passer à l'échelle quand la taille de la fédération augmente. Plus spécifiquement, ils ne parviennent pas à trouver une bonne décomposition pour les requêtes du benchmark.

(3) **Comment trouver les bons ordres de jointure pour les requêtes SPARQL property path?**, par Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli, Minh-Hoang Dang et Brice Nédelec. Dans European Semantic Web Conference, 2023.
Résumé : Les requêtes SPARQL property path offrent une manière concise d'écrire des requêtes de navigation complexes sur des graphes de connaissances. Cependant, leur évaluation sur des graphes de connaissances en ligne reste difficile, principalement en raison des fermetures transitives. En conséquence, de nombreuses requêtes property path sont interrompues par les politiques de partage équitable des SPARQL endpoints publics. Une solution pour accélérer leur exécution consiste à optimiser les ordres de jointure. Bien que le problème consistant à trouver des ordres de jointure optimaux ait déjà été largement étudié pour les requêtes

SPARQL traditionnelles, la présence des property paths biaise les approches existantes. Dans cet article, nous proposons un nouvel optimiseur de requêtes capable de mieux capturer le coût des property paths. Notre proposition s'appuie principalement sur un modèle de coût adapté aux property paths, et un estimateur de cardinalité basé sur de l'échantillonnage via des marches aléatoires. Sur le dernier benchmark de Wikidata, nous démontrons de manière empirique que notre approche trouve des ordres de jointure significativement meilleurs que Virtuoso et Blazegraph.

(4) **Comment exécuter efficament des requêtes SPARQL COUNT-DISTINCT avec la préemption Web?**, par Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli, Arnaud Grall et Thomas Minier. Dans Semantic Web Journal, 2022.
Résumé : Avoir des résultats complets lorsqu'on exécute des requêtes d'agrégation sur des SPARQL endpoints publics est difficile, principalement à cause des limitations en temps et en espace imposées par ces derniers. Bien que la préemption Web dispose d'un support côté serveur pour l'exécution des requêtes d'agrégation, les requêtes COUNT-DISTINCT restent très coûteuse à évaluer. En effet, leur évaluation avec la préemption Web nécessite de transférer une grande quantité de données. Dans cet article, nous montrons qu'il est possible de réduire considérablement la quantité de données transférées en étendant le modèle de la préemption Web avec un algorithme probabiliste appelé HyperLogLog++.

(5) **Comment exécuter des requêtes SPARQL TOP-k avec la préemption Web?**, par Julien Aimonier-Davat, Hala Skaf-Molli et Pascal Molli. Dans the 6e Workshop sur Storing, Querying and Benchmarking the Web of Data, 2022.
Résumé : L'évaluation des requêtes TOP-k sur des SPARQL endpoints public est souvent interrompue à cause des politiques de partage équitable. Bien que certaines approches, comme la préemption Web, puissent garantir des résultats complets, la performance de ces approches sur les requêtes TOP-k reste un problème. En effet, les requêtes TOP-k sont évaluées en suivant une approche "matérialise et tri" sur le client. Une telle approche nécessite de transférer tous les résultats de la requête pour au final ne garder que les k premiers. Dans cet article, (1) nous étendons le modèle de la préemption Web avec un nouvel opérateur qui permet d'évaluer des TOP-k partiels sur le serveur. Cet opérateur permet de réduire considérablement la quantité de données transferéés sur le client; (2) nous montrons que la préemption Web supporte l'early-pruning, une optimisation qui permet une réduction allant

jusqu'à 39% du temps d'exécution des requêtes TOP-k dans nos expériences sur Wikidata.

(6) **Comment exécuter des requêtes SPARQL property path avec la préemption Web?**, par Julien Aimonier-Davat, Hala Skaf-Molli et Pascal Molli. Dans European Semantic Web Conference, 2021.

Résumé : Les requêtes SPARQL property path offrent une manière concise d'écrire des requêtes de navigation complexes sur des graphes de connaissances. Cependant, l'évaluation des requêtes property path sur des graphes de connaissances en ligne, tels que DBpedia ou Wikidata, est souvent interrompue à cause des politiques de partage équitable. Pour obtenir des résultats complets, la solution consiste alors à décomposer les requêtes property path en sous-requêtes, pour lesquelles le serveur peut garantir des résultats complets. La granularité de la décomposition dépend de l'expressivité du serveur. Toutefois, quelle que soit la décomposition, cette approche peut générer un grand nombre de sous-requêtes, un transfert de données important, et finalement dégrader significativement les performances. Dans cet article, nous proposons d'étendre le modèle de la préemption Web en ajoutant un opérateur de fermeture transitive partielle côté serveur. Ensuite, nous montrons comment un client peut utiliser cet opérateur pour évaluer une requête SPARQL property path sur un graphe de connaissances en ligne, et retourner des résultats complets. Nos résultats expérimentaux démontrent que notre approche surpasse les solutions existantes, que ce soit sur le nombre d'appels HTTP, le transfert de données et le temps d'exécution des requêtes.

(7) **SaGe-Path : Évaluation progressive des requêtes SPARQL property path avec la préemption Web**, par Julien Aimonier-Davat, Hala Skaf-Molli et Pascal Molli. Dans European Semantic Web Conference, 2021.

Résumé : Les requêtes SPARQL property path permettent d'écrire des requêtes de navigation complexes sur des graphes de connaissances. Cependant, l'évaluation de ces requêtes sur des graghes de connaissances en ligne, tels que DBpedia et Wikidata, est souvent interrompue par les politiques d'utilisation équitable. SaGe-Path résoud ce problème en s'appuyant sur la préemption Web et le concept de fermeture transitive partielle. Dans le contexte d'une fermeture transitive partielle, l'exploration du graphe est limitée à une profondeur prédéfinie. Lorsque la limite de profondeur est atteinte, les noeuds les lesquelles le serveur s'est arrêté sont envoyés au client. Le client peut alors générer de nouvelles requêtes pour pour-

suivre l'exploration du graphe à partir de ces noeuds. Ainsi, SaGe-Path exécute de manière progressive, i.e., en "pay-as-you-go", les requêtes SPARQL property path. Cette démonstration montre comment des requêtes qui ne terminent pas sur le SPARQL endpoint public de Wikidata terminent en utilisant SaGe-Path. Une interface utilisateur étendue fournit une visualisation en temps réel de tous les détails internes de SaGe-Path, permettant à l'utilisateur de comprendre les overheads de l'approche et les effets des différents paramètres sur les performances. SaGe-Path démontre comment les requêtes SPARQL property path peuvent être évaluées efficacement sur des données en ligne, tout en garantissant des résultats complets.

(8) **Comment exécuter des requêtes SPARQL property path en ligne et garantir des résultats complets?**, par Julien Aimonier-Davat, Hala Skaf-Molli et Pascal Molli. Dans le 4e Workshop sur Storing, Querying and Benchmarking the Web of Data, 2020.

Résumé : Les requêtes SPARQL property path offrent un moyen concis d'écrire des requêtes de navigation complexes sur des graphes de connaissances. Cependant, l'évaluation de ces requêtes sur des graphes de connaissances en ligne, tels que DBpedia ou Wikidata, est souvent interrompue par les politiques de partage équitable. Pour obtenir des résultats complets, la solution consiste alors à décomposer les requêtes property path en sous-requêtes pour lesquelles le serveur peut garantir des résultats complets. Cependant, les décompositions existantes peuvent générer un grand nombre de sous-requêtes, un transfert de données important, et finalement dégrader significativement les performances. Dans cet article, nous proposons un nouvel algorithme capable de décomposer les requêtes SPARQL property path en un ensemble de requêtes Basic Graph Pattern (BGP). Les requêtes BGP étant garanties de terminer sur un serveur préemptif, cette décomposition garantie des résultats complets. Nos résultats expérimentaux démontrent que notre approche surpasse les approches existantes, que ce soit sur le nombre d'appels HTTP, le transfert de données et le temps d'exécution des requêtes.

(9) **FedUP: Un moteur de requête fédérée "pay-as-you-go" construit sur des marches aléatoires**, par Julien Aimonier-Davat, Brice Nédelec, Pascal Molli, Hala Skaf-Molli et Minh-Hoang Dang. Soumis à la conférence TheWebConf 2024.

Résumé : Les moteurs de requête fédérée permettent l'intégration et l'interrogation de plusieurs SPARQL endpoints comme s'il s'agissait d'un seul. Bien que les moteurs de requête fédérée évitent le déplacement des données RDF, la performance

reste un problème critique, étroitement lié à la minimisation du nombre de sous-requêtes envoyées aux SPARQL endpoints. Comme le problème de la sélection des sources est NP-difficile, les moteurs de requête fédérée existants construisent des plans de requêtes sous-optimaux, qui peuvent dégrader considérablement les performances du moteur. Dans cet article, nous proposons FedUP, un moteur de requête fédérée qui s'appuie sur des marches aléatoires pour approximer la sélection des sources. En explorant les graphes fédérés (ou leurs résumés), les marches aléatoires construisent des plans logiques capablent de capturer les relations entre les sources. En encodant cette information dans les plans, FedUP est capable de réduire considérablement le nombre de sous-requêtes envoyées aux SPARQL endpoints. Comme de nombreux résultats partagent la même combinaison de sources, les marches aléatoires convergent rapidement vers une solution qui renvoie des résultats complets et corrects. Une étude expérimentale conduite sur des benchmarks fédérés démontrent que FedUP surpasse les moteurs de requête fédérée existants, que ce soit sur la qualité de la sélection des sources, et le temps d'exécution des requêtes.

Chapter 12

Conclusion et perspectives

Pour interroger des graphes de connaissances décentralisés en ligne, nous avons identifié deux questions de recherche :

1. Comment fournir un service de requête SPARQL équitable qui garantit des résultats complets ?
2. Comment améliorer la scalabilité des moteurs de requête fédérée ?

Comment fournir un service de requête SPARQL équitable qui garantit des résultats complets? Les SPARQL endpoints publics sont soumis à une tension entre trois propriétés clés: l'équité du serveur, la complétude des résultats et le temps d'exécution des requêtes. Nous conjecturons qu'une propriété doit être sacrifiée pour garantir les deux autres propriétés. Par exemple, les politiques d'utilisation équitable appliquées par les SPARQL endpoints publics sacrifient la complétude des résultats pour garantir un service équitable et offrir de bonnes performances. Cependant, sacrifier la complétude des résulats limite drastiquement les cas d'utilisation et ainsi l'adoption du Web sémantique. Contrairement aux SPARQL endpoints, les interfaces LDF ont choisi de sacrifier la performance pour être équitables et garantir des résultats complets. L'enjeu est alors de minimiser la perte de performance tout en conservant l'équité du serveur et la garantit de retourner des résultats complets.

Parmis les moteurs de requête capable de garantir à la fois l'équité du serveur et des résultats complets, la préemption Web est l'approche qui offre les meilleures performances [92]. Malgré tout, la préemption Web n'est pas capable d'exécuter efficacement les requêtes COUNT-DISTINCT et les requêtes property path. Pour résoudre ce problème, nous avons d'abord étendu la préemption Web afin d'améliorer l'exécution des requêtes COUNT-DISTINCT :

> Julien Aimonier-Davat et al., « Online approximative SPARQL query processing for COUNT-DISTINCT queries with web preemption », *in*: *Semantic Web* 13.4 (2022), pp. 735–755, DOI: 10.3233/SW-222842

Calculer la valeur exacte d'une requête COUNT-DISTINCT nécessite de transférer une grande quantité de données [11, 53]. Pour réduire la quantité de données transférées, nous avons proposé un algorithme d'approximation pour les requêtes SPARQL COUNT-DISTINCT. Notre approche consiste à utiliser une structure de données probabilistes, i.e., HyperLogLog++, pour approcher le résultat des requêtes COUNT-DISTINCT. Le principal avantage de l'agorithme HyperLogLog++ est qu'il offre un très bon compromis entre consommation mémoire et garanti attendu sur le taux d'erreur. En intégrant cette algorithme dans le modèle de la préemption Web, nous avons démontré empiriquement que notre approche permet de calculer des requêtes COUNT-DISTINCT en ligne, avec une garantie forte sur le taux d'erreur, tout en réduisant drastiquement la quantité de données transférées par rapport aux approches existantes.

Être en mesure d'exécuter efficacement des requêtes COUNT-DISTINCT sur des graphes de connaissances en ligne ouvre la porte à de nombreux cas d'utilisation. Il est maintenant possible pour une application d'obtenir des statistiques, ou de calculer la description VoID d'un graphe de connaissances. Ces informations peuvent ensuite être utilisées dans des analyses statistiques sur les graphes de connaissances, pour faire de l'exploration de graphes, ou encore de l'optimisation de requêtes. Sans les requêtes COUNT-DISTINCT, il est par exemple impossible de répondre à certaines questions, telles que: il y a-t-il un nombre égal d'hommes et de femmes travaillant en tant que professeurs dans les universités françaises?

Ensuite, nous avons étendu la préemption Web afin d'améliorer l'exécution des requêtes property path:

> Julien Aimonier-Davat, Hala Skaf-Molli, and Pascal Molli, « Processing SPARQL Property Path Queries Online with Web Preemption », *in*: *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021*, vol. 12731, 2021, pp. 57–72.

Pour améliorer l'exécution des requêtes property path nous avons ajouté un nouvel opérateur qui permet de calculer des fermetures transitives partielles sur le serveur. Cet opérateur permet aux clients d'avoir un support côté serveur pour calculer des fermetures transitives. En utilisant ce support, un client Web peut significativement améliorer la décomposition des requêtes property path, c'est-à-dire réduire le nombre de requêtes HTTP échangées avec le serveur, et la quantité de données transférées. Nos résultats expérimentaux démontrent qu'avec un tel opérateur sur le serveur, la préemption Web est capable d'offrir de bien meilleures performances que les approches existantes, et, pour

une grande partie des requêtes property path, d'offrir des performances proches de celles d'un SPARQL endpoint.

Être en mesure d'exécuter efficacement des requêtes property path sur des graphes de connaissances en ligne rend possible plusieurs cas d'utilisation. En effet, les requêtes property path sont les seules à permettre la recherche de chemins de longueur arbitraire. Cette propriété est importante pour un utilisateur qui souhaite naviguer dans des taxonomies ou des hiérarchies complexes, ou encore chercher des connexions entre plusieurs ressources. De plus, les requêtes property path sont courrament utilisées par la communauté. Dans les logs de Wikidata, 38% sont des requêtes property path [34].

Malheureusement, même avec de très bons opérateurs côté serveur, si une requête est exécutée avec un mauvais ordre de jointure, la qualité des opérateurs n'aura qu'un impact marginale sur le temps d'exécution des requêtes [84]. Pour fournir un service efficace, il est crucial de trouver le meilleur ordre de jointure possible avant d'exécuter une requête. Si ce problème a fait l'object de nombreuses études, peu d'entres elles s'interessent aux requêtes property path. Pour combler ce vide, nous avons proposé un nouvel optimiseur pour les requêtes property path:

> Julien Aimonier-Davat et al., « Join Ordering of SPARQL Property Path Queries », *in*: *The Semantic Web - 20th International Conference, ESWC 2023, Hersonissos, Crete, Greece, May 28 - June 1, 2023, Proceedings*, vol. 13870, 2023, pp. 38–54.

Pour fonctionner, un optimiseur a besoin d'un estimateur de cardinalité, afin d'estimer le coût d'un ordre de jointure, et d'une fonction de coût, afin de comparer différents ordres de jointure. Or, aucun estimateur ne supporte les fermetures transitives, et les fonctions de coût actuelles ne sont pas capable d'estimer le coût réel d'une expression property path. Pour pallier à ces problèmes, nous avons proposé un nouvel estimateur qui utilise des marches aléatoires pour estimer la taille d'une fermeture transitive, ainsi qu'un nouvel algorithme capable d'estimer le coût réel d'un ordre de jointure avec des property paths. Cet algorithme applique des réécritures sur la requête afin de rendre le coût des property paths observable par les fonctions de coût existantes. Avec ces deux contributions, nous sommes capable de diviser par 14 le temps d'exécution des requêtes property path par rapport à Blazegraph, sur le dernier benchmark de Wikidata.

Comment améliorer la scalabilité des moteurs de requêté fédérée? Avec le benchmark FedShop [42] nous avons mis en évidence le manque de scalabilité des moteurs de requête fédérée. Actuellement, les moteurs de requête fédérée sont limités à de pe-

tites fédérations, c'est-à-dire moins de 20 SPARQL endpoints, alors que le LOD Cloud en compte plus d'un millier. Pour pallier à cette limitation, nous avons proposé FedUP, un nouveau moteur de requête fédérée. Contrairement aux moteurs de requête fédérée existants [39], FedUP génère des plans d'exécution unions-over-joins, plutôt que des plans joins-over-unions. Les plans unions-over-joins permettent de capturer des informations supplémentaires pendant l'étape de la selection des sources. Ces informations aident ensuite l'optimiseur de requête à trouver une meilleure décomposition pour les requêtes. Avec une meilleure décomposition, un moteur de requête fédérée exécute moins de jointures côté client, ce qui permet d'améliorer drastiquement les performances du moteur. Grâce à FedUP, un moteur de requête fédérée est désormais capable d'interroger efficacement des fédérations de la taille du Linked Open Data Cloud.

12.1 Perspectives

Perspectives à court terme. En combinant la préemption Web avec les différentes contributions apportées dans cette thèse, la préemption Web est maintenant capable d'exécuter efficacement l'ensemble des requêtes du benchmark WDBench [17], c'est-à-dire le workload de Wikidata. La prochaine étape consiste donc à intégrer toutes ces contributions dans un seul moteur de requête, qui sera mit à disposition de la communauté. Pour construire ce moteur de requête, nous avons décidé d'étendre le projet Apache Jena, en ajoutant à la fois un moteur de requête préemptif, et une interface pour l'échantillonnage de requête SPARQL. L'interface d'échantillonnage a déjà été publiée dans :

> Julien Aimonier-Davat et al., « RAW-JENA: Approximate Query Processing for SPARQL Endpoints », *in*: *The Semantic Web - ISWC 2023 - 22th International Semantic Web Conference, Athens, Greece, November 6-10, 2023, Proceedings*, 2023

Cette extension permet à Apache Jena d'échantillonner des requêtes SPARQL. Dans cette thèse, nous avons utilisé cette interface à deux reprises. D'un côté pour optimiser les ordres de jointure des requêtes property path. De l'autre, pour générer les plans unions-over-joins de FedUP. Toutefois, une telle interface n'est pas limitée à ces deux cas d'utilisation. Elle peut également être utilisée pour calculer des statistiques [124], des embeddings [107], ou des résumés [72] sur des graphes de connaissances en ligne. Elle peut également être utilisée pour calculer des requêtes d'agrégation en ligne [85],

ou pour explorer un graphe de connaissances [5]. Quant à la préemption Web, une partie a déjà été implémenté dans Apache Jena, i.e., tous les opérateurs core SPARQL sont supportés. Implémenter la préemption Web nécessite simplement de remplacer les opérateurs algébriques standards par des opérateurs préemptables équivalents.

Les objectifs à court terme consistent d'une part à fournir un service de requête fiable contruit sur Apache Jena, et d'autre part à mener un étude expérimentale approfondie ayant pour objectif de comparer la préemption Web tel qu'elle est aujourd'hui avec les approches LDF existantes et les SPARQL endpoints. Nous nous attendons à ce que la préemption Web offre des performances élevées dans toutes les situations, i.e., quelque soit la charge imposée au serveur.

Perspectives à long terme. Le modèle de la préemption Web peut encore être amélioré, que ce soit en ajoutant toujours plus d'opérateurs sur le serveur, ou en améliorant l'existant. Par example, au cours de cette thèse, nous avons étendu la préemption Web en ajoutant un opérateur permettant de calculer des TOP-k partiels côté serveur. Cet opérateur permet d'améliorer significativement les performances des requêtes TOP-k, notamment parce qu'il permet de faire de l'early-pruning côté serveur:

> Julien Aimonier-Davat, Hala Skaf-Molli, and Pascal Molli, « Processing SPARQL TOP-k Queries Online with Web Preemption », *in*: *Proceedings of the QuWeDa 2022: 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs co-located with 21st International Semantic Web Conference (ISWC 2022), Hangzhou, China, 23-27 October 2022*, vol. 3279, 2022, pp. 33–48.

Bien que l'early-pruning permette d'améliorer considérablement les performances des requêtes TOP-k, il serait possible d'aller plus loin en supportant l'early-termination. Ainsi, une première perspective à long terme serait d'étudier la possibilité de supporter l'early-termination côté serveur avec la préemption Web. Une seconde perspective consisterait à étudier ce qu'il se passe quand on combine plusieurs opérateurs partiels. Par exemple, quelles sont les performances que nous pouvons attendre de la préemption Web sur des requêtes qui combinent à la fois des fonctions d'agrégation et des expressions property path? Qu'en est-il pour des requêtes TOP-k construitent sur le résultat d'une fonction d'agrégation? Enfin, concernant l'ajout d'opérateurs, est-il possible d'ajouter un support côté serveur pour les requêtes property path avec des fermetures transitives imbriquées, ou pour les requêtes SPARQL avec des requêtes imbriquées?

Concernant notre algorithme d'optimisation des requêtes SPARQL, notre approche ne supporte actuellement pas les fermetures transitives imbriquées, ainsi que les opéra-

teurs MINUS, OPTIONAL, et EXISTS [10]. Une première question que l'on pourrait se poser est la suivante: comment estimer la cardinalité d'une requête MINUS ou OPTIONAL en utilisant des marches aléatoires? Une autre limitation de notre algorithme concerne les temps d'optimisation [10]. Plusieurs solutions peuvent être envisagées pour réduire le coût des marches aléatoires. Premièrement, nous pourrions définir un budget pour les marches aléatoires. Quand le budget est équisé, l'algorithme pourrait alors se tourner vers une stratégié gloutonne, et des statistiques standards, afin de garantir un temps d'optimisation borné [83]. De plus, l'effort nécessaire pour estimer un coût peut varier d'un ordre de jointure à l'autre. Adopter une stratégie adaptative pourrait permettre de mieux dépenser le budget dont nous disposons.

FedUP ouvre également un large éventail de possibilités pour des travaux futurs. Une première perspective pourrait être d'améliorer la stratégie utilisée pour réaliser les marches alétoires. Toutes les combinaisons de sources n'ayant pas la même probabilité d'être tirées, comment faire pour biaiser les marches aléatoires vers les combinaisons de sources les plus "rares"? Une telle stratégie pourrait permettre de trouver plus rapidement l'ensemble des combinaisons pertinentes pour une requête. Enfin, les plans unions-over-joins offrent la possibilité de "ranker" les combinaisons de sources. Ranker les sources pourrait permettre d'accélerer la vitesse à laquelle le moteur trouve les premiers résultats, et plus généralement, cela pourrait permettre à FedUP de réellement passer à l'échelle sur des fédérations très large.

BIBLIOGRAPHY

[1] Ibrahim Abdelaziz et al., « Lusail: A System for Querying Linked Data at Scale », *in*: *Proc. VLDB Endow.* 11.4 (2017), pp. 485–498, DOI: 10.1145/3186728.3164144.

[2] Maribel Acosta, Olaf Hartig, and Juan F. Sequeda, « Federated RDF Query Processing », *in*: *Encyclopedia of Big Data Technologies*, 2019, DOI: 10.1007/978-3-319-63962-8_228-1.

[3] Maribel Acosta et al., « ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints », *in*: *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, vol. 7031, 2011, pp. 18–34, DOI: 10.1007/978-3-642-25073-6_2.

[4] Christian Aebeloe et al., « Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns », *in*: *CoRR* abs/2002.09172 (2020).

[5] Sameer Agarwal et al., « Knowing when you're wrong: building fast and reliable approximate query processing systems », *in*: 2014, pp. 481–492, DOI: 10.1145/2588555.2593667.

[6] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish, « Efficient Management of Transitive Relationships in Large Data and Knowledge Bases », *in*: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, 1989, pp. 253–262, DOI: 10.1145/67544.66950.

[7] Julien Aimonier-Davat, Hala Skaf-Molli, and Pascal Molli, « How to Execute SPARQL Property Path Queries Online and Get Complete Results? », *in*: *Joint Proceedings of Workshops AI4LEGAL2020, NLIWOD, PROFILES 2020, QuWeDa 2020 and SEMIFORM2020 Colocated with the 19th International Semantic Web Conference (ISWC 2020), Virtual Conference, November, 2020*, vol. 2722, 2020, pp. 103–118.

[8] Julien Aimonier-Davat, Hala Skaf-Molli, and Pascal Molli, « Processing SPARQL Property Path Queries Online with Web Preemption », *in*: *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021*, vol. 12731, 2021, pp. 57–72.

[9] Julien Aimonier-Davat, Hala Skaf-Molli, and Pascal Molli, « Processing SPARQL TOP-k Queries Online with Web Preemption », *in*: *Proceedings of the QuWeDa 2022: 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs co-located with 21st International Semantic Web Conference (ISWC 2022), Hangzhou, China, 23-27 October 2022*, vol. 3279, 2022, pp. 33–48.

[10] Julien Aimonier-Davat et al., « Join Ordering of SPARQL Property Path Queries », *in*: *The Semantic Web - 20th International Conference, ESWC 2023, Hersonissos, Crete, Greece, May 28 - June 1, 2023, Proceedings*, vol. 13870, 2023, pp. 38–54.

[11] Julien Aimonier-Davat et al., « Online approximative SPARQL query processing for COUNT-DISTINCT queries with web preemption », *in*: *Semantic Web* 13.4 (2022), pp. 735–755, DOI: 10.3233/SW-222842.

[12] Julien Aimonier-Davat et al., « RAW-JENA: Approximate Query Processing for SPARQL Endpoints », *in*: *The Semantic Web - ISWC 2023 - 22th International Semantic Web Conference, Athens, Greece, November 6-10, 2023, Proceedings*, 2023.

[13] Keith Alesander et al., « Describing Linked Datasets with the VoID Vocabulary », *in*: *Recommendation W3C*, 2011.

[14] Waqas Ali et al., « A survey of RDF stores & SPARQL engines for querying knowledge graphs », *in*: *VLDB J.* 31.3 (2022), pp. 1–26, DOI: 10.1007/s00778-021-00711-3.

[15] Günes Aluç et al., « Diversified Stress Testing of RDF Data Management Systems », *in*: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, vol. 8796, 2014, pp. 197–212, DOI: 10.1007/978-3-319-11964-9_13.

[16] Thomas Anderson and Michael Dahlin, *Operating Systems: Principles and Practice*, 2nd, 2014.

[17] Renzo Angles et al., « WDBench: A Wikidata Graph Query Benchmark », *in*: *The Semantic Web - ISWC 2022 - 21st International Semantic Web Conference, Virtual Event, October 23-27, 2022, Proceedings*, vol. 13489, 2022, pp. 714–731, DOI: 10.1007/978-3-031-19433-7_41.

[18] Carlos Buil Aranda, Axel Polleres, and Jürgen Umbrich, « Strategies for Executing Federated Queries in SPARQL1.1 », *in*: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II*, vol. 8797, 2014, pp. 390–405, DOI: 10.1007/978-3-319-11915-1_25.

[19] Carlos Buil Aranda et al., « SPARQL Web-Querying Infrastructure: Ready for Action? », *in*: *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, vol. 8219, 2013, pp. 277–293, DOI: 10.1007/978-3-642-41338-4_18.

[20] Marcelo Arenas, Sebastián Conca, and Jorge Pérez, « Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard », *in*: *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, 2012, pp. 629–638, DOI: 10.1145/2187836.2187922.

[21] Diego Arroyuelo et al., « Time- and Space-Efficient Regular Path Queries », *in*: *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, 2022, pp. 3091–3105, DOI: 10.1109/ICDE53745.2022.00277.

[22] Sören Auer et al., « LODStats - An Extensible Framework for High-Performance Dataset Analytics », *in*: *Knowledge Engineering and Knowledge Management - 18th International Conference, EKAW 2012, Galway City, Ireland, October 8-12, 2012. Proceedings*, vol. 7603, 2012, pp. 353–362, DOI: 10.1007/978-3-642-33876-2_31.

[23] Amr Azzam et al., « SMART-KG: Hybrid Shipping for SPARQL Querying on the Web », *in*: *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, 2020, pp. 984–994, DOI: 10.1145/3366423.3380177.

[24] Amr Azzam et al., « WiseKG: Balanced Access to Web Knowledge Graphs », *in*: *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, 2021, pp. 1422–1434, DOI: 10.1145/3442381.3449911.

[25] Guillaume Bagan et al., « gMark: Schema-Driven Generation of Graphs and Queries », *in*: *IEEE Trans. Knowl. Data Eng.* 29.4 (2017), pp. 856–869, DOI: 10.1109/TKDE.2016.2633993.

[26] Jorge A. Baier et al., « Evaluating Navigational RDF Queries over the Web », *in*: *Proceedings of the 28th ACM Conference on Hypertext and Social Media, HT 2017, Prague, Czech Republic, July 4-7, 2017*, 2017, pp. 165–174, DOI: 10.1145/3078714.3078731.

[27] Stanislav Barton, « Designing Indexing Structure for Discovering Relationships in RDF Graphs », *in*: *Proceedings of the Dateso 2004 Annual International Workshop on DAtabases, TExts, Specifications and Objects, Desna, Czech Republic, April 14-16, 2004*, vol. 98, 2004, pp. 7–17.

[28] Tim Berners-Lee, « Linked data-design issues », *in*: (2006).

[29] Tim Berners-Lee, James Hendler, and Ora Lassila, « The semantic web », *in*: *Scientific american* 284.5 (2001), pp. 34–43.

[30] Christian Bizer, Tom Heath, and Tim Berners-Lee, « Linked Data - The Story So Far », *in*: *Int. J. Semantic Web Inf. Syst.* 5.3 (2009), pp. 1–22, DOI: 10.4018/jswis.2009081901.

[31] Christian Bizer and Andreas Schultz, « The Berlin SPARQL Benchmark », *in*: *Int. J. Semantic Web Inf. Syst.* 5.2 (2009), pp. 1–24, DOI: 10.4018/jswis.2009040101.

[32] Mike W. Blasgen et al., « The Convoy Phenomenon », *in*: *ACM SIGOPS Oper. Syst. Rev.* 13.2 (1979), pp. 20–25, DOI: 10.1145/850657.850659.

[33] Angela Bonifati, Wim Martens, and Thomas Timm, « An analytical study of large SPARQL query logs », *in*: *VLDB J.* 29.2-3 (2020), pp. 655–679, DOI: 10.1007/s00778-019-00558-9.

[34] Angela Bonifati, Wim Martens, and Thomas Timm, « Navigating the Maze of Wikidata Query Logs », *in*: *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, 2019, pp. 127–138, DOI: 10.1145/3308558.3313472.

[35] Angela Bonifati et al., *Querying Graphs*, 2018, DOI: 10.2200/S00873ED1V01Y201808DTM051.

[36] Paolo Bouquet, Chiara Ghidini, and Luciano Serafini, « A Formal Model of Queries on Interlinked RDF Graphs », *in*: *Linked Data Meets Artificial Intelligence, Papers from the 2010 AAAI Spring Symposium, Technical Report SS-10-07, Stanford, California, USA, March 22-24, 2010*, 2010.

[37] Sejla Cebiric et al., « Summarizing semantic graphs: a survey », *in*: *VLDB J.* 28.3 (2019), pp. 295–327, DOI: 10.1007/s00778-018-0528-3.

[38] Angelos Charalambidis, Antonis Troumpoukis, and Stasinos Konstantopoulos, « SemaGrow: optimizing federated SPARQL queries », *in*: *Proceedings of the 11th International Conference on Semantic Systems, SEMANTiCS 2015, Vienna, Austria, September 15-17, 2015*, 2015, pp. 121–128, DOI: 10.1145/2814864.2814886.

[39] Sijin Cheng and Olaf Hartig, « FedQPL: A Language for Logical Query Plans over Heterogeneous Federations of RDF Data Sources », *in*: *iiWAS '20: The 22nd International Conference on Information Integration and Web-based Applications & Services, Virtual Event / Chiang Mai, Thailand, November 30 - December 2, 2020*, 2020, pp. 436–445, DOI: 10.1145/3428757.3429120.

[40] Sijin Cheng and Olaf Hartig, « Towards Query Processing over Heterogeneous Federations of RDF Data Sources », *in*: *The Semantic Web: ESWC 2022 Satellite Events - Hersonissos, Crete, Greece, May 29 - June 2, 2022, Proceedings*, vol. 13384, 2022, pp. 57–62, DOI: 10.1007/978-3-031-11609-4_11.

[41] Sophie Cluet and Guido Moerkotte, « On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products », *in*: *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, vol. 893, 1995, pp. 54–67, DOI: 10.1007/3-540-58907-4_6.

[42] Minh-Hoang Dang et al., « FedShop: A Benchmark for Testing the Scalability of SPARQL Federation Engines », *in*: *The Semantic Web - ISWC 2023 - 22th International Semantic Web Conference, Athens, Greece, November 6-10, 2023, Proceedings*, 2023.

[43] Jeremy Debattista et al., « Quality Assessment of Linked Datasets Using Probabilistic Approximation », *in*: *The Semantic Web. Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings*, vol. 9088, 2015, pp. 221–236, DOI: 10.1007/978-3-319-18818-8_14.

[44] Songyun Duan et al., « Apples and oranges: a comparison of RDF benchmarks and real RDF datasets », *in*: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, 2011, pp. 145–156, DOI: 10.1145/1989323.1989340.

[45] Kemele M. Endris et al., « MULDER: Querying the Linked Data Web by Bridging RDF Molecule Templates », *in*: *Database and Expert Systems Applications - 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I*, vol. 10438, 2017, pp. 3–18, DOI: 10.1007/978-3-319-64468-4_1.

[46] Orri Erling and Ivan Mikhailov, « RDF Support in the Virtuoso DBMS », *in*: *Networked Knowledge - Networked Media - Integrating Knowledge Management, New Media Technologies and Semantic Systems*, vol. 221, 2009, pp. 7–24, DOI: 10.1007/978-3-642-02184-8_2.

[47] Cristian Estan, George Varghese, and Mike Fisk, « Bitmap algorithms for counting active flows on high speed links », *in*: *Proceedings of the 3rd ACM SIGCOMM Internet Measurement Conference, IMC 2003, Miami Beach, FL, USA, October 27-29, 2003*, 2003, pp. 153–166, DOI: 10.1145/948205.948225.

[48] Lee Feigenbaum et al., « SPARQL 1.1 Protocol », *in*: *Recommendation W3C*, 2013.

[49] Javier D. Fernández et al., « Binary RDF representation for publication and exchange (HDT) », *in*: *J. Web Semant.* 19 (2013), pp. 22–41, DOI: 10.1016/j.websem.2013.01.002.

[50] Philippe Flajolet et al., « Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm », *in*: *Discrete mathematics & theoretical computer science Proceedings* (2007).

[51] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, *Database systems - the complete book (2. ed.)* 2009, ISBN: 978-0-13-187325-4.

[52] Olaf Görlitz and Steffen Staab, « SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions », *in*: *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011*, vol. 782, 2011.

[53] Arnaud Grall et al., « Processing SPARQL Aggregate Queries with Web Preemption », *in*: *The Semantic Web - 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31-June 4, 2020, Proceedings*, vol. 12123, 2020, pp. 235–251, DOI: 10.1007/978-3-030-49461-2_14.

[54] Andrey Gubichev, « Query Processing and Optimization in Graph Databases », PhD thesis, 2015.

[55] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert, « Sparqling kleene: fast property paths in RDF-3X », *in*: *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD-/PODS 2013, New York, NY, USA, June 24, 2013*, 2013, p. 14, DOI: 10.1145/2484425.2484443.

[56] Andrey Gubichev and Thomas Neumann, « Exploiting the query structure for efficient join ordering in SPARQL queries », *in*: *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, 2014, pp. 439–450, DOI: 10.5441/002/edbt.2014.40.

[57] Andrey Gubichev and Thomas Neumann, « Path Query Processing on Very Large RDF Graphs », *in*: *Proceedings of the 14th International Workshop on the Web and Databases 2011, WebDB 2011, Athens, Greece, June 12, 2011*, 2011.

[58] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin, « LUBM: A benchmark for OWL knowledge base systems », *in*: *J. Web Semant.* 3.2-3 (2005), pp. 158–182, DOI: 10.1016/j.websem.2005.06.005.

[59] Laura M. Haas et al., « Optimizing Queries Across Diverse Data Sources », *in*: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, 1997, pp. 276–285.

[60] *Hackathon on Querying Federations of Knowledge Graphs*, 2023.

[61] Steve Harris and Andy Seaborne, « SPARQL 1.1 Query Language », *in*: *Recommendation W3C*, 2013.

[62] Andreas Harth and Sebastian Speiser, « On Completeness Classes for Query Evaluation on Linked Data », *in*: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*, 2012.

[63] Andreas Harth et al., « YARS2: A Federated Repository for Querying Graph Structured Data from the Web », *in*: *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, vol. 4825, 2007, pp. 211–224, DOI: 10.1007/978-3-540-76298-0_16.

[64] Olaf Hartig, « How Caching Improves Efficiency and Result Completeness for Querying Linked Data », *in*: *WWW2011 Workshop on Linked Data on the Web, Hyderabad, India, March 29, 2011*, vol. 813, 2011.

[65] Olaf Hartig, « SQUIN: a traversal based query execution system for the web of linked data », *in*: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, 2013, pp. 1081–1084, DOI: 10.1145/2463676.2465231.

[66] Olaf Hartig, « Zero-Knowledge Query Planning for an Iterator Implementation of Link Traversal Based Query Execution », *in*: *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I*, vol. 6643, 2011, pp. 154–169, DOI: 10.1007/978-3-642-21034-1_11.

[67] Olaf Hartig and Carlos Buil Aranda, « brTPF: Bindings-Restricted Triple Pattern Fragments (Extended Preprint) », *in*: *CoRR* abs/1608.08148 (2016).

[68] Olaf Hartig, Ian Letter, and Jorge Pérez, « A Formal Framework for Comparing Linked Data Fragments », *in*: *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, vol. 10587, 2017, pp. 364–382, DOI: 10.1007/978-3-319-68288-4_22.

[69] Ali Hasnain et al., « Extending LargeRDFBench for Multi-Source Data at Scale for SPARQL Endpoint Federation », *in*: *Emerging Topics in Semantic Technologies - ISWC 2018 Satellite Events* [*best papers from 13 of the workshops co-located with the ISWC 2018 conference*], vol. 36, 2018, pp. 203–218, DOI: 10.3233/978-1-61499-894-5-203.

[70] Ali Hasnain et al., « SPORTAL: Profiling the Content of Public SPARQL Endpoints », *in*: *Int. J. Semantic Web Inf. Syst.* 12.3 (2016), pp. 134–163, DOI: 10.4018/IJSWIS.2016070105.

[71] Lars Heling, « Quality-Driven Query Processing over Federated RDF Data Sources », *in*: *The Semantic Web: ESWC 2019 Satellite Events - ESWC 2019 Satellite Events, Portorož, Slovenia, June 2-6, 2019, Revised Selected Papers*, vol. 11762, 2019, pp. 209–219, DOI: 10.1007/978-3-030-32327-1_40.

[72] Lars Heling and Maribel Acosta, « Estimating Characteristic Sets for RDF Dataset Profiles Based on Sampling », *in*: *The Semantic Web - 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31-June 4, 2020, Proceedings*, vol. 12123, 2020, pp. 157–175, DOI: 10.1007/978-3-030-49461-2_10.

[73] Axel Hertzschuch et al., « Simplicity Done Right for Join Ordering », *in*: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*, 2021.

[74] Stefan Heule, Marc Nunkesser, and Alexander Hall, « HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm », *in*: *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, 2013, pp. 683–692, DOI: 10.1145/2452376.2452456.

[75] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman, « A survey of top-*k* query processing techniques in relational database systems », *in*: *ACM Comput. Surv.* 40.4 (2008), 11:1–11:58, DOI: 10.1145/1391729.1391730.

[76] Louis Jachiet et al., « On the Optimization of Recursive Relational Queries: Application to Graph Queries », *in*: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference* [*Portland, OR, USA*], *June 14-19, 2020*, 2020, pp. 681–697, DOI: 10.1145/3318464.3380567.

[77] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida, « A Survey of Distributed Data Aggregation Algorithms », *in*: *IEEE Commun. Surv. Tutorials* 17.1 (2015), pp. 381–404, DOI: 10.1109/COMST.2014.2354398.

[78] Riham Abdel Kader et al., « ROX: run-time optimization of XQueries », *in*: 2009, pp. 615–626, DOI: 10.1145/1559845.1559910.

[79] Mark Kaminski, Egor V. Kostylev, and Bernardo Cuenca Grau, « Query Nesting, Assignment, and Aggregation in SPARQL 1.1 », *in*: *ACM Trans. Database Syst.* 42.3 (2017), 17:1–17:46, DOI: 10.1145/3083898.

[80] Charalampos Kostopoulos et al., « KOBE: Cloud-Native Open Benchmarking Engine for Federated Query Processors », *in*: *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021, Proceedings*, vol. 12731, 2021, pp. 664–679, DOI: 10.1007/978-3-030-77385-4_40.

[81] Egor V. Kostylev et al., « SPARQL with Property Paths », *in*: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, vol. 9366, 2015, pp. 3–18, DOI: 10.1007/978-3-319-25007-6_1.

[82] Jens Lehmann et al., « DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia », *in*: *Semantic Web* 6.2 (2015), pp. 167–195, DOI: 10.3233/SW-140134.

[83] Viktor Leis et al., « Cardinality Estimation Done Right: Index-Based Join Sampling », *in*: *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[84] Viktor Leis et al., « How Good Are Query Optimizers, Really? », *in*: *Proc. VLDB Endow.* 9.3 (2015), pp. 204–215, DOI: 10.14778/2850583.2850594.

[85] Feifei Li et al., « Wander Join and XDB: Online Aggregation via Random Walks », *in*: *ACM Trans. Database Syst.* 44.1 (2019), 2:1–2:41, DOI: 10.1145/3284551.

[86] Kaiyu Li and Guoliang Li, « Approximate Query Processing: What is New and Where to Go? - A Survey on Approximate Query Processing », *in*: *Data Sci. Eng.* 3.4 (2018), pp. 379–397, DOI: 10.1007/s41019-018-0074-4.

[87] Katja Losemann and Wim Martens, « The complexity of regular expressions and property paths in SPARQL », *in*: *ACM Trans. Database Syst.* 38.4 (2013), p. 24, DOI: 10.1145/2494529.

[88] Sara Magliacane, Alessandro Bozzon, and Emanuele Della Valle, « Efficient Execution of Top-K SPARQL Queries », *in*: *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, vol. 7649, 2012, pp. 344–360, DOI: 10.1007/978-3-642-35176-1_22.

[89] Pierre Maillot et al., « IndeGx: A model and a framework for indexing RDF knowledge graphs with SPARQL-based test suits », *in*: *J. Web Semant.* 76 (2023), p. 100775, DOI: 10.1016/j.websem.2023.100775.

[90] Stanislav Malyshev et al., « Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph », *in*: *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, vol. 11137, 2018, pp. 376–394, DOI: 10.1007/978-3-030-00668-6_23.

[91] Frank Manola, Eric Miller, Brian McBride, et al., « RDF primer », *in*: *W3C recommendation* 10.*1-107* (2004), p. 6.

[92] Thomas Minier, Hala Skaf-Molli, and Pascal Molli, « SaGe: Web Preemption for Public SPARQL Query Services », *in*: *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, 2019, pp. 1268–1278, DOI: 10.1145/3308558.3313652.

[93] Gabriela Montoya, Hala Skaf-Molli, and Katja Hose, « The Odyssey Approach for Optimizing Federated SPARQL Queries », *in*: *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, vol. 10587, 2017, pp. 471–489, DOI: 10.1007/978-3-319-68288-4_28.

[94] Gabriela Montoya, Maria-Esther Vidal, and Maribel Acosta, « A Heuristic-Based Approach for Planning Federated SPARQL Queries », *in*: *Proceedings of the Third International Workshop on Consuming Linked Data, COLD 2012, Boston, MA, USA, November 12, 2012*, vol. 905, 2012.

[95] Gabriela Montoya et al., « Benchmarking Federated SPARQL Query Engines: Are Existing Testbeds Enough? », *in*: *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II*, vol. 7650, 2012, pp. 313–324, DOI: 10.1007/978-3-642-35173-0_21.

[96] Gabriela Montoya et al., « Decomposing federated queries in presence of replicated fragments », *in*: *J. Web Semant.* 42 (2017), pp. 1–18, DOI: 10.1016/j.websem.2016.12.001.

[97] Thomas Neumann and Guido Moerkotte, « Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins », *in*: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, 2011, pp. 984–994, DOI: 10.1109/ICDE.2011.5767868.

[98] Damla Oguz et al., « Federated query processing on linked data: a qualitative survey and open challenges », *in*: *Knowl. Eng. Rev.* 30.5 (2015), pp. 545–563, DOI: 10.1017/S0269888915000107.

[99] Yeonsu Park et al., « G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching », *in*: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference* [*Portland, OR, USA*], *June 14-19, 2020*, ACM, 2020, pp. 1099–1114, DOI: 10.1145/3318464.3389702.

[100] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez, « Semantics and complexity of SPARQL », *in*: *ACM Trans. Database Syst.* 34.3 (2009), 16:1–16:45, DOI: 10.1145/1567274.1567278.

[101] Axel Polleres et al., « A more decentralized vision for Linked Data », *in*: *Semantic Web* 11.1 (2020), pp. 101–113, DOI: 10.3233/SW-190380.

[102] Eric Prud'hommeaux and Carlos Buil-Aranda, « SPARQL 1.1 Federated Query », *in*: *Recommendation W3C*, 2013.

[103] Bastian Quilitz and Ulf Leser, « Querying Distributed RDF Data Sources with SPARQL », *in*: *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, vol. 5021, 2008, pp. 524–538, DOI: 10.1007/978-3-540-68234-9_39.

[104] Nur Aini Rakhmawati et al., « QFed: Query Set For Federated SPARQL Query Benchmark », *in*: *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services, Hanoi, Vietnam, December 4-6, 2014*, 2014, pp. 207–211, DOI: 10.1145/2684200.2684321.

[105] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoc, « Recursion in SPARQL », *in*: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, vol. 9366, 2015, pp. 19–35, DOI: 10.1007/978-3-319-25007-6_2.

[106] Laurens Rietveld et al., « Structural Properties as Proxy for Semantic Relevance in RDF Graph Sampling », *in*: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II*, vol. 8797, 2014, pp. 81–96, DOI: 10.1007/978-3-319-11915-1_6.

[107] Petar Ristoski and Heiko Paulheim, « RDF2Vec: RDF Graph Embeddings for Data Mining », *in*: *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I*, vol. 9981, 2016, pp. 498–514, DOI: 10.1007/978-3-319-46523-4_30.

[108] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach (4th Edition)*, 2020, ISBN: 9780134610993.

[109] Muhammad Saleem, Ali Hasnain, and Axel-Cyrille Ngonga Ngomo, « LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation », *in*: *J. Web Semant.* 48 (2018), pp. 85–125, DOI: 10.1016/j.websem.2017.12.005.

[110] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo, « HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation », *in*: *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*, vol. 8465, 2014, pp. 176–191, DOI: 10.1007/978-3-319-07443-6_13.

[111] Muhammad Saleem et al., « A fine-grained evaluation of SPARQL endpoint federation systems », *in*: *Semantic Web* 7.5 (2016), pp. 493–518, DOI: 10.3233/SW-150186.

[112] Muhammad Saleem et al., « CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation », *in*: *Proceedings of the 14th International Conference on Semantic Systems, SEMANTiCS 2018, Vienna, Austria, September 10-13, 2018*, vol. 137, 2018, pp. 163–174, DOI: 10.1016/j.procs.2018.09.016.

[113] Alexander Schätzle et al., « S2RDF: RDF Querying with SPARQL on Spark », *in*: *Proc. VLDB Endow.* 9.10 (2016), pp. 804–815, DOI: 10.14778/2977797.2977806.

[114] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim, « Adoption of the Linked Data Best Practices in Different Topical Domains », *in*: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, vol. 8796, 2014, pp. 245–260, DOI: 10.1007/978-3-319-11964-9_16.

[115] Michael Schmidt, « Foundations of SPARQL query optimization », PhD thesis, 2010.

[116] Michael Schmidt et al., « FedBench: A Benchmark Suite for Federated Semantic Data Query Processing », *in*: *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, vol. 7031, 2011, pp. 585–600, DOI: 10.1007/978-3-642-25073-6_37.

[117] Michael Schmidt et al., « SP^2Bench: A SPARQL Performance Benchmark », *in*: *Semantic Web Information Management - A Model-Based Perspective*, 2009, pp. 371–393, DOI: 10.1007/978-3-642-04329-1_16.

[118] Andreas Schwarte et al., « FedX: Optimization Techniques for Federated Query Processing on Linked Data », *in*: *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, vol. 7031, 2011, pp. 601–616, DOI: 10.1007/978-3-642-25073-6_38.

[119] Patricia G. Selinger et al., « Access Path Selection in a Relational Database Management System », *in*: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, 1979, pp. 23–34, DOI: 10.1145/582095.582099.

[120] Neha Sengupta et al., « ARROW: Approximating Reachability Using Random Walks Over Web-Scale Graphs », *in*: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, 2019, pp. 470–481, DOI: 10.1109/ICDE.2019.00049.

[121] Stephan Seufert et al., « FERRARI: Flexible and efficient reachability range assignment for graph indexing », *in*: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, 2013, pp. 1009–1020, DOI: 10.1109/ICDE.2013.6544893.

[122] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall, « The Semantic Web Revisited », *in*: *IEEE Intell. Syst.* 21.3 (2006), pp. 96–101, DOI: 10.1109/MIS.2006.62.

[123] Amritpal Singh et al., « Probabilistic data structures for big data analytics: A comprehensive review », *in*: *Knowl. Based Syst.* 188 (2020), DOI: 10.1016/j.knosys.2019.104987.

[124] Arnaud Soulet and Fabian M. Suchanek, « Anytime Large-Scale Analytics of Linked Open Data », *in*: *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I*, vol. 11778, 2019, pp. 576–592, DOI: 10.1007/978-3-030-30793-6_33.

[125] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev, « Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation », *in*: *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, 2018, pp. 1043–1052, DOI: 10.1145/3178876.3186003.

[126] Ruben Taelman et al., « Comunica: A Modular SPARQL Query Engine for the Web », *in*: *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, vol. 11137, 2018, pp. 239–255, DOI: 10.1007/978-3-030-00668-6_15.

[127] Bryan B. Thompson, Mike Personick, and Martyn Cutcher, « The Bigdata® RDF Graph Database », *in*: *Linked Data Management*, 2014, pp. 193–237, DOI: 10.1201/b16859-12.

[128] Daniel Ting, « Approximate Distinct Counts for Billions of Datasets », *in*: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, 2019, pp. 69–86, DOI: 10.1145/3299869.3319897.

[129] Giovanni Tummarello, Renaud Delbru, and Eyal Oren, « Sindice.com: Weaving the Open Linked Data », *in*: *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, vol. 4825, 2007, pp. 552–565, DOI: 10.1007/978-3-540-76298-0_40.

[130] Abhishek Udupa, Ankush Desai, and Sriram K. Rajamani, « Depth Bounded Explicit-State Model Checking », *in*: *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, vol. 6823, 2011, pp. 57–74, DOI: 10.1007/978-3-642-22306-8_5.

[131] Jürgen Umbrich et al., « Link traversal querying for a diverse Web of Data », *in*: *Semantic Web* 6.6 (2015), pp. 585–624, DOI: 10.3233/SW-140164.

[132] Pierre-Yves Vandenbussche et al., « SPARQLES: Monitoring public SPARQL endpoints », *in*: *Semantic Web* 8.6 (2017), pp. 1049–1065, DOI: 10.3233/SW-170254.

[133] Ruben Verborgh et al., « Querying Datasets on the Web with High Availability », *in*: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva*

del Garda, Italy, October 19-23, 2014. Proceedings, Part I, vol. 8796, 2014, pp. 180–196, DOI: 10.1007/978-3-319-11964-9_12.

[134] Ruben Verborgh et al., « Triple Pattern Fragments: A low-cost knowledge graph interface for the Web », *in*: *J. Web Semant.* 37-38 (2016), pp. 184–206, DOI: 10.1016/j.websem.2016.03.003.

[135] Sarisht Wadhwa et al., « Efficiently Answering Regular Simple Path Queries on Large Labeled Networks », *in*: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, 2019, pp. 1463–1480, DOI: 10.1145/3299869.3319882.

[136] Andreas Wagner, Veli Bicer, and Thanh Tran, « Pay-as-you-go Approximate Join Top-k Processing for the Web of Data », *in*: *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*, vol. 8465, 2014, pp. 130–145, DOI: 10.1007/978-3-319-07443-6_10.

[137] Andreas Wagner et al., « Top-k Linked Data Query Processing », *in*: *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*, vol. 7295, 2012, pp. 56–71, DOI: 10.1007/978-3-642-30284-8_11.

[138] Dong Wang, Lei Zou, and Dongyan Zhao, « Top-k queries on RDF graphs », *in*: *Inf. Sci.* 316 (2015), pp. 201–217, DOI: 10.1016/j.ins.2015.04.032.

[139] Xin Wang, Thanassis Tiropanis, and Hugh C. Davis, « LHD: Optimising Linked Data Query Processing Using Parallelisation », *in*: *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013*, vol. 996, 2013.

[140] Yuxiang Wang et al., « Approximate and Interactive Processing of Aggregate Queries on Knowledge Graphs: A Demonstration », *in*: *Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022*, 2022, pp. 5034–5038, DOI: 10.1145/3511808.3557158.

[141] Kyu-Young Whang, Brad T. Vander Zanden, and Howard M. Taylor, « A Linear-Time Probabilistic Counting Algorithm for Database Applications », *in*: *ACM Trans. Database Syst.* 15.2 (1990), pp. 208–229, DOI: 10.1145/78922.78925.

[142] Gang Wu et al., « System Pi: A Native RDF Repository Based on the Hypergraph Representation for RDF Data Model », *in*: *J. Comput. Sci. Technol.* 24.4 (2009), pp. 652–664, DOI: 10.1007/s11390-009-9265-9.

[143] Qingjun Xiao et al., « Hyper-Compact Virtual Estimators for Big Network Data Based on Register Sharing », *in*: *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Portland, OR, USA, June 15-19, 2015*, 2015, pp. 417–428, DOI: 10.1145/2745844.2745870.

[144] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz, « Evaluation of SPARQL Property Paths via Recursive SQL », *in*: *Proceedings of the 7th Alberto Mendelzon International Workshop on Foundations of Data Management, Puebla/Cholula, Mexico, May 21-23, 2013*, vol. 1087, 2013.

[145] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz, « Query Planning for Evaluating SPARQL Property Paths », *in*: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 1875–1889, DOI: 10.1145/2882903.2882944.

[146] Weipeng P. Yan and Per-Åke Larson, « Eager Aggregation and Lazy Aggregation », *in*: *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, 1995, pp. 345–357.

[147] Shengqi Yang et al., « Fast top-k search in knowledge graphs », *in*: *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, 2016, pp. 990–1001, DOI: 10.1109/ICDE.2016.7498307.

[148] Shima Zahmatkesh et al., « Towards a Top-K SPARQL Query Benchmark », *in*: *Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014*, vol. 1272, 2014, pp. 349–352.

COLLEGE ED MATHEMATIQUES
DOCTORAL STIC
PAYS DE LA LOIRE

Nantes Université

Titre : Comment interroger un graphe de connaissance décentralisé en ligne ?

Mot clés : Web sémantique, serveurs SPARQL publics, préemption Web, optimisation, moteur de requête fédérée

Résumé :

Dans l'état actuel du Web sémantique, le développement d'applications intelligentes reposant sur un graphe de connaissances décentralisé accessible en ligne demeure un rêve lointain. D'une part, les SPARQL endpoints publics font face à de graves problèmes d'accessibilité. De l'autre, les moteurs de requête fédérée, qui permettent aux utilisateurs d'interroger de manière transparente plusieurs SPARQL endpoints, ne passent pas à l'échelle. Ainsi, des milliards de triples RDF sont disponibles mais restent inaccessibles. Bien que diverses solutions aient été proposées, elles présentent toutes des problèmes de performance importants. Dans cette thèse, nous proposons différentes solutions pour surmonter ces limitations. Pour commencer, nous étendons le modèle de la préemption Web afin de fournir un service de requête SPARQL public, fiable, et efficace. Ensuite, nous proposons un nouvel algorithme d'optimisation pour les requêtes SPARQL conjonctives qui contiennent des filtres et des expressions property path. Enfin, nous proposons un nouveau moteur de requête fédérée capable d'interroger efficacement une fédération de serveurs SPARQL de la taille du Linked Open Data Cloud.

Title: Querying Online Decentralized Knowledge Graphs

Keywords: Semantic Web, Public SPARQL Servers, Web Preemption, Join Ordering, Federation Engine

Abstract: In the current state of the Semantic Web, developing intelligent applications on top of a live queryable decentralized knowledge graph remains a distant dream. On the one hand, public SPARQL endpoints face serious availability problems. On the other hand, federation engines, which allow users to query multiple endpoints as a single one, do not scale when the number of sources increases. As a result, billions of RDF triples are available but not accessible. Various solutions have been proposed to solve these problems, but existing solutions all present significant performance issues. In this thesis, we propose solutions to overcome these limitations. We start by extending the Web preemption model with new preemptable operators. In doing so, we significantly improve the performance of frequently used classes of queries. Next, we propose a new optimization algorithm for conjunctive SPARQL queries with filters and property paths. Finally, we introduce a new federation engine that can handle federations of the size of the Linked Open Data Cloud.