Web Preemption for Querying the Linked Open Data

Thomas Minier

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Thèse de doctorat

Thomas Minier

Université de Nantes
Comue Université Bretagne Loire
École Doctorale N°601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Web Preemption for Querying the Linked Open Data

Thèse présentée et soutenue à Nantes, le 10 Novembre 2020
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes

Composition du Jury

Présidente :	**M^me^ Pascale KUNTZ**	Professeure, Université de Nantes
Rapporteurs :	**Mr Fabien GANDON**	Directeur de Recherche, INRIA Sophia Antipolis
	Mr Ruben VERBORGH	Professeur, IDLab - Ghent University
Examinatrice :	**M^me^ Fatiha SAÏS**	Professeure, Université Paris-Sud
Directeur de thèse :	**Mr Pascal MOLLI**	Professeur, Université de Nantes
Co-directrice de thèse :	**M^me^ Hala SKAF-MOLLI**	Maître de conférence/HDR, Université de Nantes

Rules and responsibilities: these are the ties that bind us. We do what we do, because of who we are. If we did otherwise, we would not be ourselves. I will do what I have to do. And I will do what I must.

Neil Gaiman, *The Sandman: Book of Dreams*, 1996

Sometimes scientists change their minds. New developments cause a rethink. If this bothers you, consider how much damage is being done to the world by people for whom new developments do not cause a rethink.

Terry Pratchett, *The Science Of Discworld*, 1999

Remerciements

Je remercie Pascal Molli et Hala Skaf pour m'avoir offert la possibilité de réaliser cette thèse et de m'avoir accompagné durant ces trois années. Je remercie Matthieu Perrin, Achour Mostefaoui, Patricia Serrano Alvarado et Emmanuel Desmontils, dont les avis extérieurs m'ont aidé à garder un esprit critique sur mon travail. Je remercie Jérémie Bourdon, pour ses précieux conseils tout au long de ma thèse et pour m'avoir fait découvrir l'informatique il y a huit années de cela. Je remercie Annie Boileau et Élodie Guidon, pour leur efficacité et leur professionnalisme sans faille. Et plus généralement, je remercie tous les autres membres de l'équipe GDD et du LS2N pour leur accueil.

Je remercie Arnaud, Benjamin et Thibault, pour leur soutien, le sacro-saint café de 12h30 et le traditionnel Berlin du mois. Je remercie également Brice et Luiz, pour leur solidarité et leurs conseils avisés.

Je remercie Alicia, Pierre, Lenny, Théo, Kevin et Joachim pour les moments partagés depuis ces folles années de Master jusqu'à aujourd'hui, et surtout pour m'aider à garder les pieds sur terre et ne pas perdre la vue d'ensemble sur ce surprenant domaine qu'est l'informatique.

Je remercie chaleureusement la folle bande des Ukulélés, Rémi, Myriam, Thomas, Guillaume, Philippe et Alexandre, pour les méfaits passés, présents et futurs.

Je remercie tout particulièrement Pauline, toujours à l'écoute, et Nicolas, toujours fidèle au poste, pour avoir relu le présent manuscrit et fourni de nombreux retours, scientifiques comme stylistiques.

Enfin, et surtout, je remercie mes parents Pierre et Hélène, mon frère Hugo, mes grand-parents Anne-Françoise, Paul, Sylvie et Yves, et mes oncles Arnaud, Julien et Vincent, pour leur soutien depuis toujours.

Contents

Chapter 1

Introduction

1.1 Motivation

The original Web of documents [12] is based on three key pillars. The first is the Unique Resource Identifier (URI) which associates each resource on the Web to unique identifiers. The most common form of an URI is a Uniform Resource Locator (URL), *e.g.*, `https://www.univ-nantes.fr` is the URL of the welcome page of the University of Nantes website. The second pillar is the Hypertext Transfer Protocol (HTTP), which allows accessing a URL, and either the document identified by the URI, or an error code. For example, the famous *404 error code* indicates that the requested resource does not exist. The third and final fundamental notion is the Hypertext Markup Language (HTML), a hypertext format used to describe the content of web pages.

However, conventional HTML documents are not expressive enough to unambiguously describe entities and their properties across the World Wide Web [15]. To this end, the Semantic Web [13] extends the classic Web of documents to create a *Web of Data*, where text documents are replaced by structured data that can be automatically processed by computers. It re-uses the first two fundamental notions (URLs and the HTTP protocol) from the Web of documents, but replaces the use of HTML documents by Resource Description Framework (RDF) documents [18]. The core structure of the RDF data model [18] is a set of *RDF triples*. A triple is composed of a *subject*, a *predicate*, and an *object*, and asserts facts about entites. Figure 1.1 shows some RDF triples extracted from DBpedia that asserts facts about the entity Neil Gaiman [74]: he is an author, he is named "Neil Gaiman" and he wrote the books Amercian Gods and Good Omens.

Tim Berners-Lee [11] proposed four principles to publish RDF documents so that they become part of a single global data space: the Linked Open Data (LOD). These principles are as follows:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix dbo:  <http://dbpedia.org/ontology/>
@prefix dbr:  <http://dbpedia.org/resource/>
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix wikidata: <https://www.wikidata.org/entity/Q210059>

dbr:Neil_Gaiman rdf:type dbo:Person .
dbr:Neil_Gaiman foaf:name "Neil Gaiman" .
dbr:Neil_Gaiman owl:sameAs wikidata:Q210059 .

dbr:American_Gods dbo:author dbr:Neil_Gaiman .

dbr:Good_Omens dbo:author dbr:Neil_Gaiman .
```

Figure 1.1: Turtle representation of RDF data about Neil Gaiman, retrieved from DBpedia [9].

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

Following these principles, data providers published billions of RDF documents over the Web [15, 61]. Most of them are hosted as static RDF files and can be accessed by *dereferencing* the URI of entities. For example, by dereferencing the URI `http://dbpedia.org/resource/Neil_Gaiman`, we can access the RDF document that describes all the facts asserted about Neil Gaiman in DBpedia [1]. These documents are linked using predicates, to associate entities across distinct RDF documents. For example, in Figure 1.1, the `http://www.w3.org/2002/07/owl#sameAs` predicate connects two entities (`dbo:Neil_Gaiman` and `wikidata:Q1052459`) that both represent the same information (the author Neil Gaiman), but distributed across two distinct RDF documents (DBpedia and Wikidata [2]). This interlinked Web of Data constitutes a global space of Linked Data documents called the *Linked Open Data cloud* (LOD cloud). Figure 1.2 shows the state of the LOD cloud as a diagram [3]. As of March 2020, it contains 1255 datasets with 16174 links.

However, hosting public RDF files does not allow data consumers to easily answer complex questions using the LOD. For example, if you want to list all

[1] https://wiki.dbpedia.org

[2] https://www.wikidata.org

[3] The LOD cloud diagram is maintained by John P. McCrae for the Insight Centre for Data Analytics, and can be consulted at https://lod-cloud.net

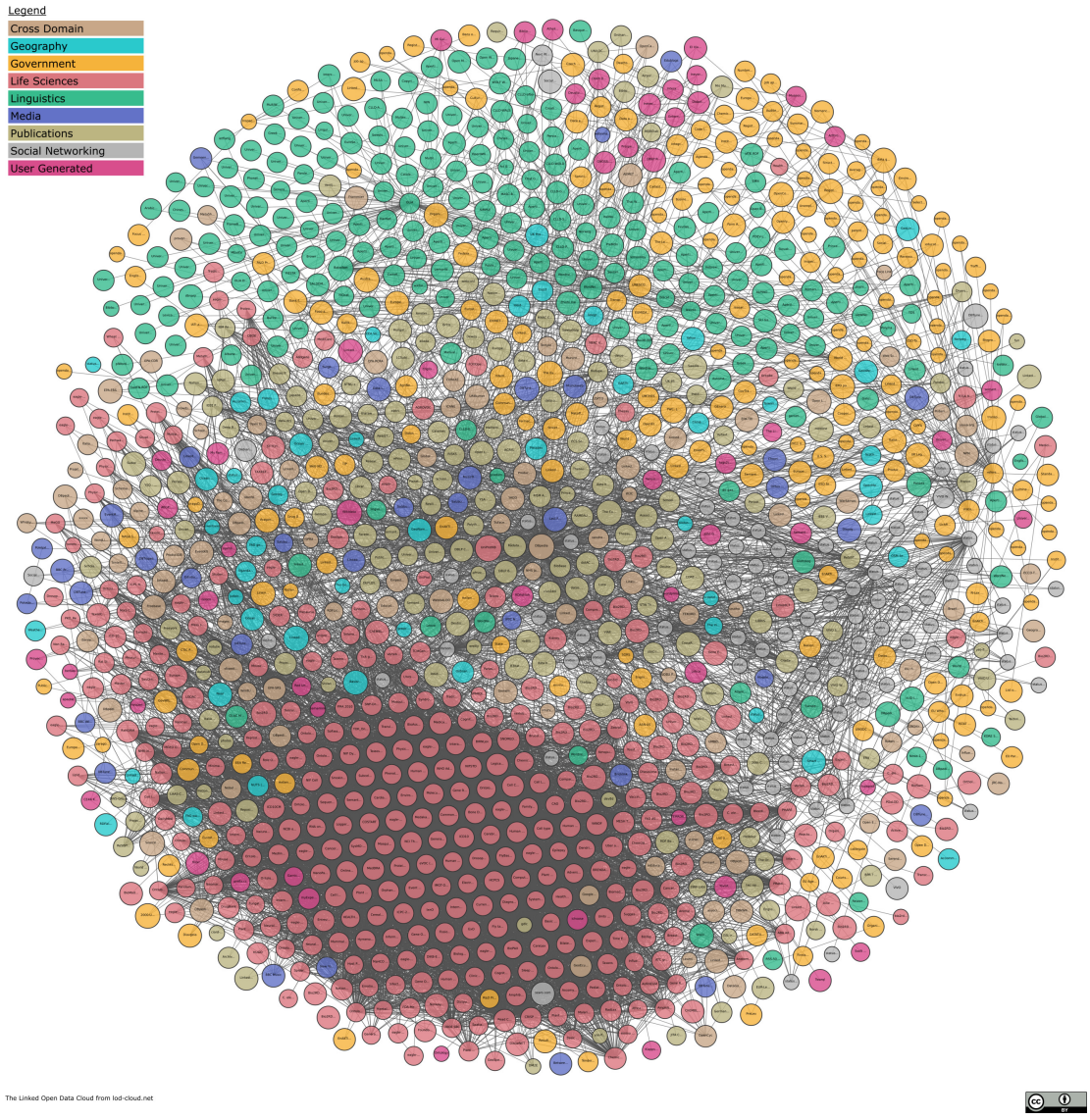


Figure 1.2: The LOD cloud diagram, as of March 2019.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX dbr:  <http://dbpedia.org/resource/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?author ?book WHERE {
  ?author rdf:type dbo:Person .
  ?author foaf:name ?name .
  ?book dbo:author ?author .
}
```

Figure 1.3: A SPARQL query that finds all authors and their books in the DBpedia dataset.

actors and their movies, you need to *crawl* the LOD to download every document that contains relevant informations, which is very costly considering the current size of the LOD. Alternatively, a collection of RDF documents can be hosted in a dedicated database and exposed using a *SPARQL endpoint*. A SPARQL endpoint follows the SPARQL protocol [4], which "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". Hence, users can send SPARQL queries to the endpoint, which are SQL-like queries that allow users to query RDF documents with an expressive query language. For example, the DBpedia SPARQL endpoint [5] enables data consumers to execute queries on the DBpedia dataset, composed of billions of facts retrieved from Wikipedia [9]. Figure 1.3 shows a SPARQL query that finds all authors and their books in the DBpedia dataset.

Formally, the SPARQL protocol defines a SPARQL query execution model with two actors: 1) A *client* (a human or a software) which wants to execute a SPARQL query. 2) A *SPARQL endpoint*, which is an HTTP server that evaluates SPARQL queries over an *RDF dataset*. To execute a SPARQL query, a web client sends a SPARQL query to the server through an HTTP GET or POST request. The SPARQL endpoint will evaluate it against a default RDF Graph, whose URI is set by the client, and then returns the SPARQL query results to the client.

Public SPARQL query servers, like SPARQL endpoints, are very important for the Web of Data, as they enable Web developers to write applications that re-use the billions of RDF triples available in the LOD cloud. For example, both the DBpedia [9] and Wikidata [73] SPARQL endpoints are used as backends by multilingual question answering platforms [22, 69] and chatbots [8]. However,

[4] https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321
[5] http://dbpedia.org/sparql

providing a public service that allows anyone to execute any SPARQL query at any time is still an open issue [6]. As public SPARQL query services can suffer from an unpredictable load of arbitrary SPARQL queries, the challenge is to ensure that the service remains *available* despite variation in terms of the arrival rate of *requests* and *resources* required to process queries. Providers often refer to this challenge as "maintaining a fair usage policy".

To achieve a fair usage policy, most SPARQL endpoints configure quotas on their servers to prevent the convoy effect and ensure a fair sharing of resources among end-users. If a SPARQL query exceeds one of these quotas, the server rejects it, and the web client receives an error. There exists many type of quotas, almost one set per data provider. Most commonly, data providers restrict: 1) The execution time of SPARQL queries, 2) The number of results sent per query, and 3) The arrival rate of queries per IP. For example, the DBpedia SPARQL endpoint does not allow queries to run for more than 120 seconds or return more than 10000 results [6]. Additionally, it only allows 50 parallel connections and 100 HTTP requests per second per IP address. The Wikidata SPARQL endpoint enforces even harder limitations, as it restricts SPARQL query execution time to 60 seconds with only one connection allowed per IP address [7].

Some of these quotas are not disturbing for web clients, *e.g.*, if the arrival rate is restricted, then clients only need to try again later. However, limiting the execution time of SPARQL queries causes some queries to deliver only *partial results*. To illustrate, consider the execution of the SPARQL query of Figure 1.3. When executed with a local SPARQL endpoint without any quotas nor limitations, the total number of results for this query is 35 215. However, when executed against the public DBpedia SPARQL endpoint, we found only 10 000 results out of 35 215 [8], because the query exceeded the maximum number of results to send per query. *Delivering partial results is a severe limitation for a public SPARQL service*, as it prevents users from building Web applications based on SPARQL endpoints, especially in the domain of Life Sciences [58].

However, without a quota policy, SPARQL endpoints can suffer from *convoy effect* [16], because they execute queries using a First-Come First-Served (FCFS) execution policy [25]: queries are executed in their order of arrival at the endpoint. Thus, one long-running query can occupy the server resources and prevents others from executing. Convoy effects can severely deteriorate query execution performance in terms of 1) *average waiting time*: the average time spent by queries in the server's waiting queue, 2) *average time for first results*: the average time between sending the query and receiving the first bytes of query results, and 3) *average query*

[6] http://wiki.dbpedia.org/public-sparql-endpoint

[7] https://www.mediawiki.org/wiki/Wikidata_Query_Service/User_Manual#Query_limits

[8] All results were obtained on DBpedia version 2016-04.

completion time: the average time between sending the query to the server and receiving the last bytes of query results.

In the current state of the LOD, we face a dilemma. On one hand, we can build a responsive public SPARQL query server using a quota policy, but it can provide incomplete results. On the other hand, we can build a server that always delivers complete results, but can become unresponsive due to convoy effects. Neither solution is suitable for developping Web applications over the Web of data, as web developers need responsive SPARQL query servers that always deliver complete results.

1.2 Research outline and contributions

In this thesis, we choose to tackle the issue of **building public SPARQL query servers that allow any data consumer to execute any SPARQL query with complete results**. Existing approaches address this issue by decomposing SPARQL queries into subqueries that can be run under the quotas and produce complete results [7]. Finding such decomposition is hard in the general case, as quotas can be different from one server to another, both in terms of values and nature [7]. The Linked Data Fragments (LDF) approach [38, 72] tackles this issue by restricting the SPARQL operators supported by the server. For example, in the Triple Pattern Fragments (TPF) approach [72], a TPF server only evaluates triple patterns. However, current LDF approaches produce a large number of subqueries and substantial data transfer.

In this work, we choose to follow a different path from the state of the art, as we believe that the issue related to quotas is not interrupting a query but the impossibility for the client to *resume* the query execution afterward. In consequence, we propose **Web Preemption**, a new execution model for public SPARQL query servers, inspired by the works on *time-sharing* [5]. Web preemption is the capacity of a Web server to suspend a running query after a time quantum with the intention to resume it later. When suspended, the state S_i of the query is returned to the Web client. Then, the client can resume query execution by sending S_i back to the Web server. Compared to existing query execution models, Web Preemption ensures *a fair allocation of Web server resources across queries, a better average query completion time per query, and a better time for first results.*

However, our new model adds an overhead for the Web server to suspend the running query and resume the next waiting query. Consequently, the next scientific challenge here is to keep this overhead marginal, whatever the running SPARQL queries, to ensure top query execution performance. At this end, we propose **SaGe**, a preemptive SPARQL query engine that implements the Web Preemption model. We define a set of **preemptable query operators** for which

we bound the complexity of suspending and resuming of these operations, both in time and space. They allow us to build a preemptive Web server that supports a large fragment of the SPARQL query language. As not all SPARQL operations exhibit a low preemption overhead, we split the SaGe query engine into two components. First, a **SaGe SPARQL query server** uses our set of preemptable operators for executing a large fragment of the SPARQL query language under the Web Preemption model. Second, a **SaGe Smart Web Client**, executes the remaining SPARQL operations client-side and applies optimizations to reduce the data transfer. By combining the two components, we allow any user to evaluate any SPARQL 1.1 query under the Web Preemption model.

Finally, we provide a complete experimental study of the SaGe SPARQL query engine, client and server, compared to existing approaches used for hosting public SPARQL services. Experimental results demonstrate that SaGe outperforms existing approaches by several orders of magnitude in terms of the average total query execution time and the time for the first results.

1.3 Thesis overview

This thesis is constructed as follows.

Chapter 2 presents the background notions and the related work of this thesis. It provides background on the RDF data model, the formal semantics of the SPARQL query language, the main techniques for storing RDF data and the various methods for processing SPARQL query over RDF datasets hosted in a centralized database. It also provides related work on convoy effects, quotas policy and techniques to circumvent them.

Chapter 3 presents and motivates the Web preemption model, our first contribution of this thesis. It formalizes the Web Preemption model and presents the challenge related to its implementation.

Chapter 4 presents the SaGe preemptive SPARQL query engine, our second contribution of this thesis. It defines a set of *preemptable physical SPARQL operators*, used to evaluate SPARQL queries under the Web Preemption model. For each preemptable operator, we prove its space and time complexity. We also introduce the SaGe Smart Web Client, which comes with its own set of challenges regarding query performance, and presents the experimental study that compares SaGe to state of the art public SPARQL query servers.

Chapter 5 concludes this thesis and outlines perspective works.

Annex A provides a summary of this thesis in French.

1.4 Publications

This thesis is primarily based on the following publication.

> **SaGe: Web Preemption for Public SPARQL Query services**, by Thomas Minier, Hala Skaf-Molli and Pascal Molli. In Proceedings of the 2019 World Wide Web Conference (WWW'19), San Francisco, United States, May 13-17 (2019).
> **Abstract:** To provide stable and responsive public SPARQL query services, data providers enforce quotas on server usage. Queries which exceed these quotas are interrupted and deliver partial results. Such interruption is not an issue if it is possible to resume queries execution afterward. Unfortunately, there is no preemption model for the Web that allows for suspending and resuming SPARQL queries. In this paper, we propose SaGe: a SPARQL query engine based on Web preemption. SaGe allows SPARQL queries to be suspended by the Web server after a fixed time quantum and resumed upon client request. Web preemption is tractable only if its cost in time is negligible compared to the time quantum. The challenge is to support the full SPARQL query language while keeping the cost of preemption negligible. Experimental results demonstrate that SaGe outperforms existing SPARQL query processing approaches by several orders of magnitude in term of the average total query execution time and the time for first results.

During my thesis, I have also contributed to the following publications that do not directly relate to its core topic, presented in reverse chronological order follows.

1. **Processing SPARQL Aggregates Queries with Web Preemption**, by Arnaud Grall, Thomas Minier, Hala Skaf-Molli and Pascal Molli. In Proceedings of the 17th European Semantic Web Conference (ESWC 2020), Heraklion, Greece (2020).
   Abstract: Executing aggregate queries on the web of data allows to compute useful statistics ranging from the number of properties per class in a dataset to the average life of famous scientists per country. However, processing aggregate queries on public SPARQL endpoints is challenging, mainly due to quotas enforcement that prevents queries to deliver complete results. Existing distributed query engines allow to go beyond quota limitations, but their data transfer and execution times are clearly prohibitive when processing aggregate queries. Following the web preemption model, we define a new preemptable aggregation operator that allows to suspend and resume aggregate queries. Web preemption allows to continue query execution beyond quota limits and server-side aggregation drastically reduces data transfer and execution time

of aggregate queries. Experimental results demonstrate that our approach outperforms existing approaches by several orders of magnitude in terms of execution time and the amount of transferred data.

2. **Intelligent clients for replicated Triple Pattern Fragments**, by Thomas Minier, Hala Skaf-Molli, Pascal Molli and Maria-Esther Vidal. In Proceedings of the 15th European Semantic Web Conference (ESWC 2018), 400-414, Heraklion, Greece (2018).
   Abstract: Following the Triple Pattern Fragments (TPF) approach, intelligent clients are able to improve the availability of the Linked Data. However, data availability is still limited by the availability of TPF servers. Although some existing TPF servers belonging to different organizations already replicate the same datasets, existing intelligent clients are not able to take advantage of replicated data to provide fault tolerance and load-balancing. In this paper, we propose Ulysses, an intelligent TPF client that takes advantage of replicated datasets to provide fault tolerance and load-balancing. By reducing the load on a server, Ulysses improves the overall Linked Data availability and reduces data hosting cost for organizations. Ulysses relies on an adaptive client-side load-balancer and a cost-model to distribute the load among heterogeneous replicated TPF servers. Experimentations demonstrate that Ulysses reduces the load of TPF servers, tolerates failures and improves queries execution time in case of heavy loads on servers.

3. **Ulysses: an Intelligent client for replicated Triple Pattern Fragments**, by Thomas Minier, Hala Skaf-Molli, Pascal Molli and Maria-Esther Vidal. In The Semantic Web: ESWC 2018 Satellite Events, Heraklion, Greece (2018).
   Abstract: Ulysses is an intelligent TPF client that takes advantage of replicated datasets to distribute the load of SPARQL query processing and provides fault-tolerance. By reducing the load on a TPF server, Ulysses improves the Linked Data availability and distributes the financial costs of queries execution among data providers. This demonstration presents the Ulysses web client and shows how users can run SPARQL queries in their browsers against TPF servers hosting replicated data. It also provides various visualizations that show in real-time how Ulysses performs the actual load distribution and adapts to network conditions during SPARQL query processing.

4. **Parallelizing Federated SPARQL Queries in Presence of Replicated Data**, by Thomas Minier, Gabriela Montoya, Hala Skaf-Molli and Pascal Molli. In ESWC 2017 - Revised Selected Papers. Springer, Cham, 2017. p. 181-196 (2017).

Abstract: Federated query engines have been enhanced to exploit new data localities created by replicated data, e.g., Fedra. However, existing replication aware federated query engines mainly focus on pruning sources during the source selection and query decomposition in order to reduce intermediate results thanks to data locality. In this paper, we implement a replication-aware parallel join operator: Pen. This operator can be used to exploit replicated data during query execution. For existing replication-aware federated query engines, this operator exploits replicated data to parallelize the execution of joins and reduce execution time. For Triple Pattern Fragment (TPF) clients, this operator exploits the availability of several TPF servers exposing the same dataset to share the load among the servers. We implemented Pen in the federated query engine FedX with the replicated-aware source selection Fedra and in the reference TPF client. We empirically evaluated the performance of engines extended with the Pen operator and the experimental results suggest that our extensions outperform the existing approaches in terms of execution time and balance of load among the servers, respectively.

5. **PeNeLoop: Parallelizing federated SPARQL queries in presence of replicated fragments**, by Thomas Minier, Gabriela Montoya, Hala Skaf-Molli and Pascal Molli. In Querying the Web of Data (QuWeDa 2017) Workshop, co-located with 14th ESWC 2017 (Vol. 1870, pp. 37-50) (2017).
Abstract: Replicating data fragments in Linked Data improves data availability and performances of federated query engines. Existing replication aware federated query engines mainly focus on source selection and query decomposition in order to prune redundant sources and reduce intermediate results thanks to data locality. In this paper, we extend replication-aware federated query engines with a replication-aware parallel join operator: PeNeLoop. PeNeLoop exploits redundant sources to parallelize the join operator and reduce execution time. We implemented PeNeLoop in the federated query engine FedX with the replicated-aware source selection Fedra and we empirically evaluated the performance of FedX+Fedra+PeNeLoop. Experimental results suggest that FedX+Fedra+PeNeLoop outperforms FedX+Fedra in terms of execution time while preserving answer completeness.

Chapter 2

Background and Related Works

In this chapter, we first present basic notions on the RDF data model and the SPARQL query language. Then, we review background on data management techniques for storing and querying RDF data under centralized approaches, *i.e.*, data are stored in a single database system and queried using the SPARQL query language. Finally, we present related works on the thesis scientific challenge. We review techniques for circumventing quotas on public SPARQL endpoints, alternatives to SPARQL endpoints for hosting RDF data and we elaborate on the convoy effect and its roots in the Operating System community.

2.1 Background

2.1.1 The RDF data model

The core structure of the RDF data model [18] is a set of RDF triples, each composed of a *subject*, a *predicate*, and an *object*. A collection of such triples is called an *RDF graph*, and a set of RDF graphs is called an *RDF dataset*. Within such a dataset, each graph is identified by a URI (except for the default graph), hence talk about *named RDF graph*. In a way, an RDF graph is a directed and labeled multi-graph of nodes, where each RDF triple corresponds to a directed edge where the predicate is the label, the subject is the source node, and the object is the target node. There are three types of nodes in a graph: URIs, *literals*, and *blank nodes*. A URI or a Literal denotes that a specific resource, or *entity*, exists in the Web of Data: the resource indicated by a URI is called its referent, and the resource denoted by a literal is called its literal value. Unlike URIs and literals, blank nodes do not identify specific resources. Statements involving blank nodes say that something with the given relationships exists, without explicitly naming it.

```
@prefix foaf:    <http://xmlns.com/foaf/0.1/>
@prefix dbo:     <http://dbpedia.org/ontology/>
@prefix dbr:     <http://dbpedia.org/resource/>
@prefix owl:     <http://www.w3.org/2002/07/owl#>
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
@prefix wikidata: <https://www.wikidata.org/entity/Q210059>

dbr:Neil_Gaiman rdf:type dbo:Person .
dbr:Neil_Gaiman foaf:name "Neil Gaiman" .
dbr:Neil_Gaiman dbo:author dbr:American_Gods .
dbr:Neil_Gaiman dbo:author dbr:Good_Omens .
dbr:Neil_Gaiman owl:sameAs wikidata:Q210059 .

dbr:American_Gods dbo:author dbr:Neil_Gaiman .
dbr:American_Gods rdfs:label "American Gods"@en .

dbr:Good_Omens dbo:author dbr:Neil_Gaiman .
dbr:Good_Omens dbo:author dbr:Terry_Pratchett .
dbr:Good_Omens rdfs:label "Good Omens"@en .
```

Figure 2.1: Turtle representation of RDF data about Neil Gaiman and the book Good Omens and American Gods, retrieved from DBpedia.

```
<http://dbpedia.org/resource/Neil_Gaiman> # Subject
<http://dbpedia.org/ontology/author> # Predicate
<http://dbpedia.org/resource/Good_Omens> # Object
```

Asserting an RDF triple indicates that *a relation, shown by the predicate, holds between the resources denoted by the subject and object.* For example, the above RDF triple represents the fact that Neil Gaiman wrote the book Good Omens. By *dereferencing* the URI http://dbpedia.org/resource/Neil_Gaiman, we can access the RDF document that describes all the facts asserted about Neil Gaiman in the DBpedia dataset [1]. Figure 2.1 shows an extract of this document in Turtle notation. Figure 2.2 shows the same data but represented as a directed, labeled RDF graph.

[1] https://wiki.dbpedia.org/

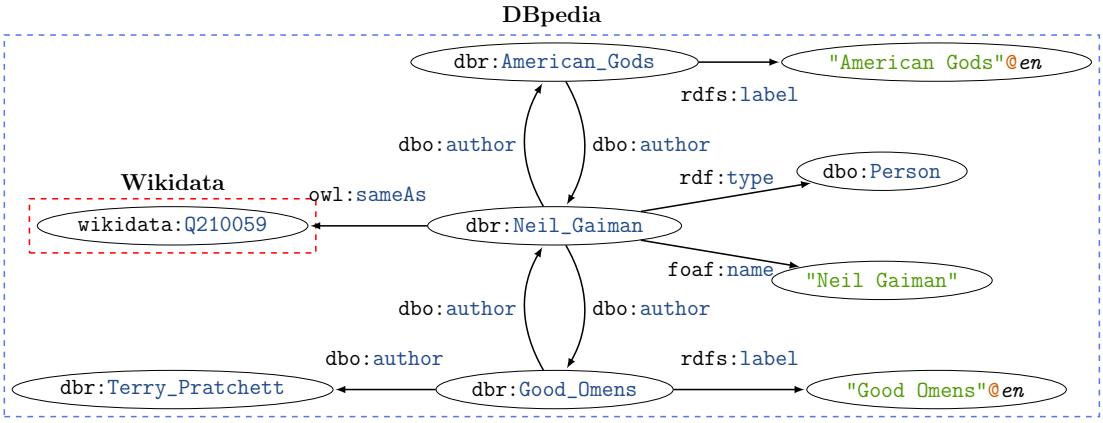


Figure 2.2: Graph representation of the facts about Neil Gaiman and the books Good Omens and American Gods.

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/> .
PREFIX dbo:   <http://dbpedia.org/ontology/> .
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

SELECT ?author ?book_title WHERE {
  ?author rdf:type dbo:Person .
  ?author dbo:author ?book .
  ?book rdfs:label ?book_title .
}
```

Figure 2.3: A SPARQL query that finds all authors with their names and the titles of their books.

2.1.2 The SPARQL query language

The SPARQL query language [32] allows data consumers to query RDF documents using more complex logic, similar to SQL queries for relational databases. The SPARQL query Q_1 from Figure 2.3 finds all authors with their names and the titles of their books. If we execute Q_1 on the set of RDF triples shown in Figure 2.1, we obtain the following results.

?author	?book_title
<http://dbpedia.org/resource/Neil_Gaiman>	"American Gods"@en
<http://dbpedia.org/resource/Neil_Gaiman>	"Good Omens"@en

The semantic of the SPARQL query language was formalized by Perez et al. in

[56], and further extended by Schmidt et al. in [62]. Let $\mathbb{U}, \mathbb{V}, \mathbb{L}$ be disjoint infinite sets of URIs, SPARQL variables, and literals, respectively. A SPARQL query is composed of *SPARQL Graph patterns expressions*, defined recursively as follows:

- A triple pattern $tp \in (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{L} \cup \mathbb{V})$ is a graph pattern.
- If P_1 and P_2 are graph patterns, then the expressions P_1 AND P_2, P_1 UNION P_2 and P_1 OPT P_2 are graph patterns.
- If P is a graph pattern and R is a SPARQL built-in conditions, then P FILTER R is a graph pattern.

An expression P_1 AND P_2 AND ... AND P_N is often called a *Basic Graph Pattern* (BGP), *i.e.*, a conjunctive query between several graph patterns. A SPARQL built-in condition is constructed using elements of the set $\mathbb{U} \cup \mathbb{L} \cup \mathbb{V}$ and constants, logical connectives $(\neq, \wedge, \vee)$, inequality symbols $(<, \leq, >, \geq)$, the equality symbol $(=)$ and unary predicates [2].

The evaluation of a SPARQL graph pattern P against an RDF dataset $\mathcal{D}$, denoted $[\![P]\!]_{\mathcal{D}}$, produces a set of *solution mappings*. A solution mapping μ is a partial function that maps a set of variables to an RDF term. The *domain* of μ, denoted $dom(\mu)$, is the set of variables on which μ is defined. Two mappings μ and μ' are *compatible*, denoted $\mu \sim \mu'$, if $\forall x \in dom(\mu) \cap dom(\mu'), \mu(x) = \mu'(x)$. Furthermore, $var(tp)$ denotes all variables in a triple pattern tp and $\mu(tp)$ denotes the RDF triple obtained by replacing the variables in tp according to μ.

Definition 1 gives the algebra of the SPARQL query language and Definition 2 gives the evaluation semantic of SPARQL graph patterns expressions.

Definition 1 (SPARQL algebra [56, 62])**.** *Let Ω_1, Ω_2 be sets of solutions mappings, $\mathcal{R}$ a filter condition, and V a finite set of variables. The expressions of the SPARQL algebra are defined as follows:*

$$
\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\} \\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\} \\
\Omega_1 \setminus \Omega_2 &= \{\mu_1 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \nsim \mu_2\} \\
\Omega_1 \mathbin{⟕} \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2) \\
\sigma_{\mathcal{R}}(\Omega_1) &= \{\mu \mid \mu \in \Omega_1, \mu \models R\} \\
\pi_V(\Omega_1) &= \{\mu \mid \exists \mu' : \mu \cup \mu' \in \Omega_1 \wedge dom(\mu) \subseteq V \\
&\qquad \wedge dom(\mu') \cap V = \emptyset\}
\end{aligned}
$$

where $\models$ refers to built-in boolean functions defined in [56].

[2] https://www.w3.org/TR/sparql11-query/#SparqlOps

Definition 2 (SPARQL semantics [56, 62])**.** *Let $\mathcal{D}$ be an RDF dataset, tp a triple pattern, $\mathcal{R}$ a FILTER expression, V a finite set of variables, and P_1, P_2 SPARQL graphs patterns. The evaluation of a graph pattern over $\mathcal{D}$, denoted $[\![\cdot]\!]_{\mathcal{D}}$, is defined recursively as follows:*

$$
\begin{aligned}
[\![tp]\!]_{\mathcal{D}} &= \{\mu \mid dom(\mu) = var(tp) \wedge \mu(tp) \in \mathcal{D}\} \\
[\![P_1 \ AND \ P_2]\!]_{\mathcal{D}} &= [\![P_1]\!]_{\mathcal{D}} \bowtie [\![P_2]\!]_{\mathcal{D}} \\
[\![P_1 \ UNION \ P_2]\!]_{\mathcal{D}} &= [\![P_1]\!]_{\mathcal{D}} \cup [\![P_2]\!]_{\mathcal{D}} \\
[\![P_1 \ OPT \ P_2]\!]_{\mathcal{D}} &= [\![P_1]\!]_{\mathcal{D}} \rtimes [\![P_2]\!]_{\mathcal{D}} \\
[\![P_1 \ FILTER \ \mathcal{R}]\!]_{\mathcal{D}} &= \sigma_{\mathcal{R}}([\![P_1]\!]_{\mathcal{D}}) \\
[\![\ SELECT \ V \ \ WHERE \ P]\!]_{\mathcal{D}} &= \pi_V([\![P]\!]_{\mathcal{D}})
\end{aligned}
$$

To illustrate, consider again the SPARQL query from Figure 2.3. In the formal SPARQL semantic, the query is composed of a projection and a Basic Graph Pattern with the following four triple patterns:

$$
\begin{aligned}
tp_1 &= \texttt{?author rdf:type dbo:Person} \\
tp_2 &= \texttt{?author dbo:author ?book} \\
tp_3 &= \texttt{?book rdf:type dbo:Book} \\
tp_4 &= \texttt{?book rdfs:label ?book\_title}
\end{aligned}
$$

So, according to Definition 2, the SPARQL query is formally expressed as

$$
\begin{aligned}
Q = \ & \text{SELECT } \{\texttt{?author}, \texttt{?book\_title}\} \text{ WHERE } \{tp_1 \text{ AND } tp_2 \text{ AND } tp_3 \text{ AND } tp_4\} \\
= \ & \pi_{\texttt{?author},\texttt{?book\_title}}(tp_1 \bowtie tp_2 \bowtie tp_3 \bowtie tp4)
\end{aligned}
$$

2.1.3 Linked Data Management

From a conceptual point of view, we can distinguish two distinct approaches for storing RDF data. The *relational-based* approach, which views the RDF dataset as a set of RDF triples and considers it as a particular case of the relational data model. The main advantage of this approach is that one can re-use all techniques for indexing, storing, and querying data developed for relational databases [26, 27]. This approach also highlights the similarities between SPARQL and SQL query languages. On the other hand, the *graph-based* approach considers an RDF graph through its graph-based representation and leverages graph-based algorithms to perform query processing. However, they are widely adopted for building public SPARQL endpoints. Consequently, in this thesis, we focus on the relational-based approach because they allow building SPARQL query servers with complete SPARQL query processing capabilities. We now review the main approaches for storing RDF data under the relational-bases approach.

Subject	Predicate	Object
dbr:American_Gods	dbo:author	dbr:Neil_Gaiman
dbr:American_Gods	rdfs:label	"American Gods"@en
dbr:Good_Omens	dbo:author	dbr:Neil_Gaiman
dbr:Good_Omens	dbo:author	dbr:Terry_Pratchett
dbr:Good_Omens	rdfs:label	"Good Omens"@en
dbr:Neil_Gaiman	dbo:author	dbr:American_Gods
dbr:Neil_Gaiman	dbo:author	dbr:Good_Omens
dbr:Neil_Gaiman	foaf:name	"Neil Gaiman"
dbr:Neil_Gaiman	rdf:type	dbo:Person
dbr:Neil_Gaiman	owl:sameAs	wikidata:Q210059

(a) SPO Index

Predicate	Object	Subject
dbo:author	dbr:American_Gods	dbr:Neil_Gaiman
dbo:author	dbr:Good_Omens	dbr:Neil_Gaiman
dbo:author	dbr:Neil_Gaiman	dbr:American_Gods
dbo:author	dbr:Neil_Gaiman	dbr:Good_Omens
dbo:author	dbr:Terry_Pratchett	dbr:Good_Omens
foaf:name	"Neil Gaiman"	dbr:Neil_Gaiman
owl:sameAs	wikidata:Q210059	dbr:Neil_Gaiman
rdf:type	dbo:Person	dbr:Neil_Gaiman
rdfs:label	"American Gods"@en	dbr:American_Gods
rdfs:label	"Good Omens"@en	dbr:Good_Omens

(b) POS Index

Object	Subject	Predicate
"American Gods"@en	dbr:American_Gods	rdfs:label
dbo:Person	dbr:Neil_Gaiman	rdf:type
dbr:American_Gods	dbr:Neil_Gaiman	dbo:author
dbr:Good_Omens	dbr:Neil_Gaiman	dbo:author
dbr:Neil_Gaiman	dbr:American_Gods	dbo:author
dbr:Neil_Gaiman	dbr:Good_Omens	dbo:author
dbr:Terry_Pratchett	dbr:Good_Omens	dbo:author
"Good Omens"@en	dbr:Good_Omens	rdfs:label
"Neil Gaiman"	dbr:Neil_Gaiman	foaf:name
wikidata:Q210059	dbr:Neil_Gaiman	owl:sameAs

(c) OSP Index

Figure 2.4: Storing RDF data from Figure 2.1 using a triple-store with three clustered indexes: SPO, POS and OSP.

Triple stores [23, 30, 34, 53, 75] put all RDF triples into a single table with three attributes: *subject*, *predicate*, and *object*. An optional fourth column, *graph*, can be used to record the named RDF graph that contains each triple of the dataset.

As in classical relational databases, query processing over a triple store can speed up drastically by defining *indexes* over the table attributes. Thus, many triple stores use a set of indexes on several combinations of the subject (S), predicate (P) and object (O) attributes. One of the first approaches [48] rely on the following fives distinct B+-trees [17]: 1) one of each attribute S, P and O, 2) a joint index on S and P, 3) a joint index on S and P and a separate index on O, 4) a joint index on P and O, 5) a joint index on S, P, and O. This indexing schema proved to be very efficient for executing simple lookup queries, *i.e.*, scanning for a triple pattern, and queries with few joins.

To further speed up query processing, many approaches rely on *clustered indexes*, *i.e.*, indexes that store multiples copies of the same data but in different orders. Such indexes allow us to look for random triple patterns in the RDF Graph efficiently and to use fast merge joins for executing queries with a large number of joins. Hexastore [75] relies on six B^+-trees on all possible permutations of S, P, and O: SPO, SOP, POS, PSO, OSP, and OPS. Figure 2.4 shows how Hexastore stores the

RDF data from Figure 2.1. For the sake of simplicity, we only show the SPO, POS, and OSP indexes. To evaluate a triple pattern *tp* against such indexes, Hexastore looks at the position of variables in *tp*, selects the appropriate index, and starts an *Index Scan* [26, 27] over the index. For example, for scanning a triple pattern tp = `?s rdf:type ?o`, Hexastore selects either the index POS or PSO, locates the record with key p = `rdf:type` and iterates over all matching RDF triples. Additionally, as Hexastore stores RDF triples in every possible order, the query engine can use very fast *merge joins* [26, 27] to resolve joins between triple patterns. The theoretical upper bound on space consumption in Hexastore is equals to five times the size of the triple table, and much lower in practice, due to the skewed nature of RDF graphs.

RDF-3X [53] follows a similar aggressive indexing technique with the same indexes as Hexastore. Additionally, it also indexes all subsets of the (S, P, O) set, adding nine new partial indexes called *aggregated indexes*. These indexes associate the parts of RDF triples with their number of occurrences in the RDF dataset, which allows the query engine to estimate the cardinality of triple patterns. For example, the aggregated index SP stores, for each key $\langle s, p \rangle$, the number of RDF triples whose subject is s and predicate is p ($|\{s \mid \forall \langle s, p, o' \rangle \in \mathcal{G}\}|$). The drawback of such aggressive indexing techniques is a significant space overhead, as clustered indexes stores multiples copies of the original data. To mitigate this overhead, RDF-3X relies on a *delta compression* of RDF triples in the B^+-trees leaves. Since two adjacent triples are likely to share the same prefix, then only the difference (*delta*) between triples are stored, instead of the full triples.

Hexastore and RDF-3X are RDF databases that employ very aggressive indexing schemes. On the opposite, Virtuoso [23] uses a partial indexing scheme, with only two full indexes on PSOG and POGS, where G is the URI of the graph to which the triple belongs. Besides, Virtuoso also stores three partial indexes SP, OP, and GS, which allows it to evaluate any triple pattern by combining partial indexes.

Property tables rely on the *vertical partitioning* technique [1]. They group RDF triples per predicate value, and all triples within the same group are stored in a separate table, named after the predicate. Each table contains two attributes: *subject* and *predicate*, where at least the subject attribute is indexed using a (un)clustered B^+-tree. The kind of approach can be implemented very efficiently in column-based stores [43, 64]. Figure 2.5 shows how the RDF data from Figure 2.1 are stored using property tables.

Clustered-property tables [31, 65, 76] extend the concept of property tables and groups RDF triples into classes based on the co-occurrence of sets of predicates in the dataset. Each table stores the triple that corresponds to a class, with

Subject	Object
dbr:American_Gods	dbr:Neil_Gaiman
dbr:Good_Omens	dbr:Neil_Gaiman
dbr:Good_Omens	dbr:Terry_Pratchett
dbr:Neil_Gaiman	dbr:American_Gods
dbr:Neil_Gaiman	dbr:Good_Omens

(a) Predicate dbo:author

Subject	Object
dbr:Neil_Gaiman	"Neil Gaiman"

(b) Predicate foaf:name

Subject	Object
dbr:Neil_Gaiman	wikidata:Q210059

(c) Predicate owl:sameAs

Subject	Object
dbr:Neil_Gaiman	dbo:Person

(d) Predicate rdf:type

Subject	Object
dbr:American_Gods	"American Gods"@en
dbr:Good_Omens	"Good Omens"@en

(e) Predicate rdfs:label

Figure 2.5: Storing RDF data from Figure 2.1 using property tables.

attributes named after the predicates. The classes are usually computed using a clustering algorithm or by an expert.

A major drawback of both property tables and clustered-property tables is that they offer poor performance when executing SPARQL queries that contain unbounded predicates. In this case, the query engine must scan all tables in the database to find matches, and then all results need to be re-combined using joins or unions. Additionally, each time we add a new predicate to the dataset, the RDF database needs to recompute the classes and the data partitions, as the content of some classes might evolve, which is very expensive for large RDF datasets. Due to these hard limitations, property tables and clustered-property tables have not become popular in RDF databases. Consequently, in this thesis, we focus on Linked Data Management using Triple stores.

2.1.4 SPARQL query processing in Triple Stores

Execution of SPARQL queries over RDF data is a process called *SPARQL query processing*, with a vast literature. A good survey on the whole field is in [33]. In this thesis, we focus on SPARQL query processing over Triple stores, as they are the most dominant RDF database systems for public SPARQL query servers.

Given a SPARQL query, a SPARQL query engine constructs a query execution plan and executes it on the RDF database to produce the query results. This

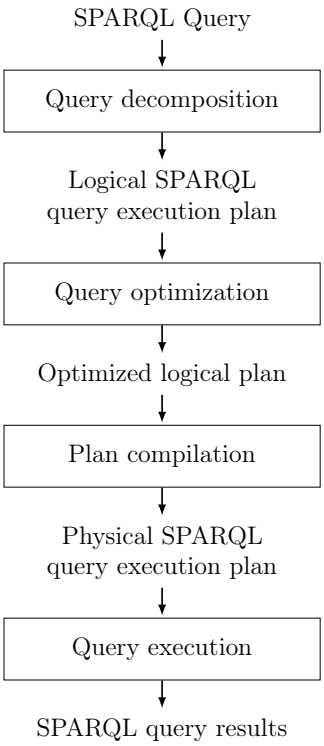


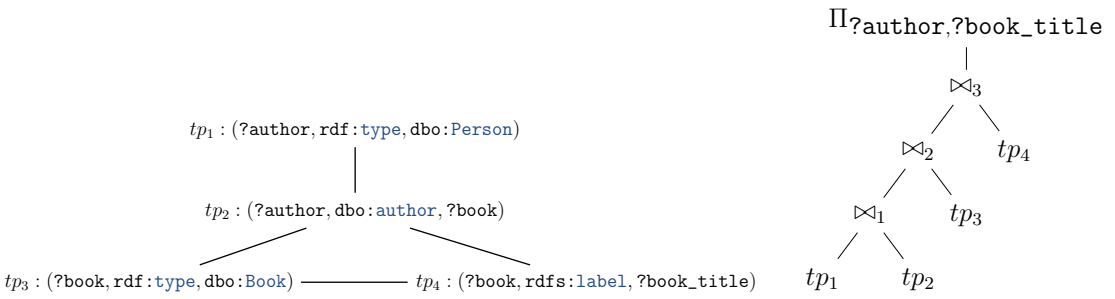
Figure 2.6: Overview of SPARQL query processing

process is called the *optimize-then-execute* paradigm [27]. It allows the query engine to translates a program written in a high-level specification (the SPARQL query language) into an optimized low-level computer program (the query execution plan), and the execution of this program produces the query results. Figure 2.6 summarizes the overall process of SPARQL query processing.

2.1.4.1 Query decomposition

The first layer of SPARQL query processing parses the query and decomposes it into an intermediate representation. First, the query is parsed and *normalized* into an Abstract Syntax Tree (AST), more suitable for manipulation by the query engine. The AST is *semantically analyzed* so that incorrect queries are detected and rejected as early as possible.

Next, the query engine translates the AST into a representation called a *join query graph* [67], where each Basic Graph pattern is expanded into a set of triple patterns. The triple patterns are then turned into the edge of the join query graph, where two nodes of the graph are connected with an edge if the corresponding triple patterns share at least one SPARQL variable. This representation corresponds



(a) Join query graph

(b) Logical SPARQL query execution plan

Figure 2.7: Join query graph and logical query execution plan of SPARQL query Q_1 from Figure 2.3.

to the traditional conjunctive query graph from the relational query optimization model, where nodes are relations and edges are join predicates. Figure 2.7a shows the join query graph corresponding to the SPARQL query from Figure 2.3.

Next, given the join query graph, the SPARQL query engine constructs its algebraic representation [56], called a *join tree*, composed of logical scan and join operators. The join tree is a binary tree whose leaves are the triple patterns, and inner nodes are joins. We distinguish two important class of join tree: 1) *left-deep (linear) tree*, where every join has one of its triple patterns has input, and 2) *bushy trees*, where no restriction applies.

Finally, the query engine completes the join tree with the other operators from the SPARQL query, *e.g.*, OPTIONAL, UNION, or aggregations. These disjunctive parts of the query are treated as nested SPARQL subqueries, in that there inner basic graph patterns are translated and optimized separately. Then, the query engine combines these sub-parts with the join tree, producing a *logical SPARQL query execution plan*. Figure 2.7b shows the logical execution plan generated for the SPARQL query from Figure 2.3.

2.1.4.2 Query optimization

The goal of the second layer of SPARQL query processing is to find an execution strategy for the SPARQL query which is close to optimal [37, 62], as finding the optimal execution plan is computationally intractable. Query optimization consists of finding the "best" ordering of operators in the query plan to minimize a *cost function*. This function, often defined in terms of time units, refers to computing resources such as disk space, disk I/Os, buffer space, CPU cost, etc. The most simple optimization techniques are based on SPARQL normal forms [62]. For

example, the *join distribution* technique allows creating bushy trees from joins between UNION clauses. The *FILTER push-down* technique consists of splitting FILTER conditions into the conjunction of conditions and pushing these conditions as deep as possible towards the leaves of the join tree. These techniques are easy to apply, as they are *syntactic*: they are based on rules that transform the plan based on its shape, so they can be used by any SPARQL query engine.

One of the most critical optimizations done by the SPARQL query engine is called *join ordering* [53, 67]. It consists of re-ordering the join sequence of triple patterns in each join tree to minimize the *selectivity* of each join operator. The cardinality of a triple pattern is the number of RDF triples that matches this pattern. Let c_1, c_2 be the cardinality of two triple patterns tp_1, tp_2, then the selectivity of the join $tp_1 \bowtie tp_2$ is defined as $S(tp_1 \bowtie tp_2) = \frac{|tp_1 \bowtie tp_2|}{c_1 \times c_2}$. Typically, selectivities are *estimated* by the query optimizer using precomputed statistics [52, 67]. These selectivities provide a way to optimize the cardinality of intermediate results generated by a join tree. The state of the art techniques that solves this problem are based on dynamic programming algorithms [50, 53].

2.1.4.3 Building the physical SPARQL query execution plan

Once the optimized logical SPARQL query execution plan is produced, the SPARQL query optimizer invokes the last layer of SPARQL query processing to compile the logical plan into a *physical SPARQL query execution plan*. This plan is quite literally the software that needs to be executed to produce query results.

To produce this plan, the query engine performs a tree-traversal of the logical plan and, for each logical operation, selects a *physical query operator* to implement it. While the logical algebra used in the logical plan is the same for all SPARQL queries, the physical algebra is *system-specific*: different databases may implement the same logical operators using different physical operators [27]. For example, a triple store can evaluate triple patterns using *scan operators* on the available indexes. In contrast, a database that relies on vertical partitioning will use operators specific to its partition layout.

As several physical operators can implement the same logical operator, the query engine relies on a *cost-model* to select the most efficient implementation for each operator. For example, the Merge join algorithm is more performant if both join inputs produce results sorted on the join variable. Figure 2.8 shows a SPARQL physical query execution plan compiled from the logical plan of Figure 2.7b. In this plan, triple patterns are evaluated using *Index Scans*, which scans B$^+$-trees indexes to produces solutions mappings from relevant RDF triples. A Merge join implements the first join in the plan, as the solutions mappings produced by both its inputs are sorted on the `?author` join variable. Remaining joins are done using

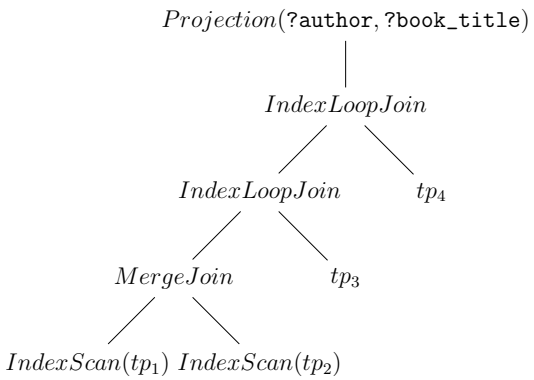


Figure 2.8: Physical query execution compiled from the logical plan of Figure 2.7b.

Iterator	`Open()`	`GetNext()`	`Close()`
Scan	Get a pointer to the first matching RDF triple	Read the next matching RDF triple	Release the pointer
Filter	Open input	Read items from the input until one matches the SPARQL expression	Close input
Sort	Read all items from the input, materialize them into a table and sort the table	Read the next item from the sorted table	Cleanup the sorted table and close input

Table 2.1: Examples of Iterators for SPARQL query processing

Index Loop joins [21], a variant of the well-known loop join algorithm that leverages the indexes of our triple store to speed-up query processing. Finally, a Projection operator executes the logical $\pi_{\texttt{?author,?book\_title}}$ operation and produce the query solutions for the SPARQL query Q_1.

Another choice made by the query engine during the plan compilation phase is *how physical operators communicate between them.* For example, given a query with two joins in sequence, how do the results of the first join are passed to the second one? The simplest method is to rely on temporary files to exchange intermediate results between operators. However, for queries with a large number of intermediate results, this method adds significant overhead in terms of data storage during query execution [27]. Alternatively, many query engines follow the *iterator model* [28], where operators are connected as a pipeline and communicate in sequence. Each time an operator needs to produce some value, it pulls solutions

from its predecessor(s) in the pipeline, computes the value, and then forward it to the next operator in the pipeline. Operators implemented with this model are called *iterators* and formalized in Definition 3. Table 2.1 shows some examples of physical operators implemented as iterators.

Definition 3 (Iterator)**.** *A physical operator that implements the iterator model supports the following operations:*

- *`Open()` which initialize the datastrucures used by the operator.*
- *`GetNext()` which produce an intermediate result.*
- *`Close()` which cleanup any datastrucure used by the operator.*

Implementing physical operators as a pipeline of iterators provide several properties for query execution: 1) operators are implemented as individual units, independent of each other, 2) the whole query execution is executed by a single process in a *coroutine-fashion*, 3) operators produce one item at a time upon request, 4) intermediate results are passed directly between operators without the need for intermediate files, which makes this model very efficient in terms of memory costs, and 5) a pipeline of iterators naturally implements any shape of query plan.

However, a pipeline of iterators is quite inefficient in terms of CPU consumption, because `GetNext` functions will be called recursively for producing every single solution mappings. As each call to `GetNext` will cause a jump in the heap memory, queries with a large number of intermediate results can cause many round-trips in the CPU, deteriorating performance and generating poor code locality, which reduces optimizations from modern CPUs [51]. Nonetheless, this approach remains the most used implementation for physical query execution in most databases.

2.2 Related Works

2.2.1 Decomposing SPARQL queries to avoid quotas

A first approach to circumvent quotas is to decompose a SPARQL query that is interrupted by some quotas into a set of subqueries whose execution completes under the same quotas. In [7], Buil Aranda et al. study such type of evaluation strategies for federated SPARQL queries, but they can be generalized to non-federated queries. The authors identify six distinct rewriting strategies, ranging from simple pagination to more complex query rewriting techniques.

Pagination (SYMHASHP) The most straightforward technique to circumvent the limitations imposed by quotas is to *paginate* the query results using the `LIMIT`, `OFFSET`, and `ORDER BY` solution modifiers. We first sort all query results and then retrieve them by batches of fixed sizes. The union of the solutions of all

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?actor ?name WHERE {
  {
    SERVICE <http://dbpedia.org/sparql> {
      SELECT ?actor ?name WHERE {
        ?actor a dbo:Actor. ?actor rdfs:label ?name.
      } ORDER BY ?actor ?name LIMIT 10000
    }
  } UNION {
    SERVICE <http://dbpedia.org/sparql> {
      SELECT ?actor ?name WHERE {
        ?actor a dbo:Actor. ?actor rdfs:label ?name.
      } ORDER BY ?actor ?name OFFSET 10000 LIMIT 10000
    }
  }
}
```

Figure 2.9: A paginated SPARQL query

subqueries yields the final query results. The sorting phase is mandatory, because the SPARQL endpoint may not produce the results in the same order between each query. Figure 2.9 shows an example of a SPARQL query paginated with the SYMHASHP strategy, where we fetch query results per page of 10 000 results. However, while this approach is feasible, it is not applicable in general, as executing several consecutive `ORDER BY` queries might actually be quite expensive on the remote server and could trigger resource limits nonetheless. For example, the Virtuoso SPARQL endpoint [23], used as the backend for the DBpedia SPARQL endpoint, limits the amount of memory per query when sorting results. We can avoid such limitations if the endpoint already delivers sorted results, but it makes the approach very dependent on the actual server implementation.

Symmetric Hash Join (SYMHASH) This strategy is applicable only for the evaluation of Basic Graph patterns, *i.e.*, joins of triple/graph patterns. To evaluate a SPARQL query $Q = P_1 \bowtie P_2$, the client executes P_1 and P_2 on the server using two distinct requests, stores their results into two hash tables (one per subquery), and then joins the results using the Symmetric Hash Join algorithm [77]. This strategy can be generalized for joining an arbitrary number of graph patterns. However, this join algorithm is expensive if the intermediate result sets are much larger then the join result size [77] ($|[\![P_1]\!]_{\mathcal{D}}| + |[\![P_2]\!]_{\mathcal{D}}| > |[\![P_1 \bowtie P_2]\!]_{\mathcal{D}}|$), and thus

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?actor ?name WHERE {
  ?actor rdfs:label ?name.
  VALUES ?actor { dbp:David_Tennant dbp:Michael_Sheen }
}
```

Figure 2.10: A SPARQL query that "ships" solution bindings to perform a join server-side, following the `VALUES`, `FILTER` and `UNION` strategies [7].

may deteriorate query processing performance. This strategy also requires that the evaluations of both P_1 and P_2 deliver complete results under quotas, or the join results will be incomplete.

Nested Loop Join (NESTED) Similar to SYMHASH, the join between two graph patterns is done on the client-side using the Nested Loop Join algorithm [26]. Given a query $P = P_1 \bowtie P_2$, the client evaluates P_1, and then for each $\mu \in [\![P_1]\!]_{\mathcal{D}}$, the client evaluates $\mu(P_2)$ using the server and produces the final join results. However, this approach faces three significant issues. First, the Nested Loop join algorithm is efficient only if the selectivity of the left join input is less than the selectivity of the right input [27] ($|[\![P_1]\!]_{\mathcal{D}}| < |[\![P_2]\!]_{\mathcal{D}}|$). So, like the SYMHASH strategy, it can deteriorate query performance if the algorithm inputs do not meet these requirements. Second, this strategy sends an HTTP request to the server for each solution binding of P_1. If P_1 has a high number of solutions, then the NESTED strategy can lead to denial of service attacks or trigger quotas on request-rate. For example, the DBpedia SPARQL endpoint does not allow more than 100 requests per second per IP address, with an initial burst of 120 requests. Finally, only a subclass of SPARQL queries called *strongly bounded SPARQL queries* are guaranteed to produce complete results with this technique, as shown in [7].

`VALUES`, `FILTER`, and `UNION` strategies The last three techniques improve on the NESTED strategy to reduce the data transferred between client and server. Instead of sending one HTTP request per solution of $[\![P_1]\!]_{\mathcal{D}}$, the client "ships" a set of solutions, using the `VALUES` operator, to perform the join between P_1 and P_2 server-side. A similar technique, called *Bound join*, is used by the FedX federated SPARQL query engine [63]. Given $P = P_1 \bowtie P_2$, the client sends a SPARQL query to the server to compute $\Omega = [\![P_1]\!]_{\mathcal{D}}$, and then sends a second query P_2 `VALUES` Ω to perform the join server-side. Figure 2.10 shows a SPARQL query that performs data shipping. It computes the join between P_2 = `?actor rdfs:label ?name`

and the set of solutions $\Omega = \{\{\texttt{?actor} \mapsto \texttt{dbp:David\_Tennant}\}, \{\texttt{?actor} \mapsto \texttt{dbp:Michael\_Sheen}\}\}$ to finds the label of the actors David Tennant and Michael Sheen. As the `VALUES` operator is not widely available on current public SPARQL endpoints [6], the authors propose equivalent strategies using the `FILTER` and `UNION` SPARQL operators. However, as before, this strategy requires that both queries sent to the server produce complete results under quotas.

Synthesis In summary, all of these strategies suffer from the same significant drawbacks. First, they need to know the server configuration in terms of quotas. This information might be unavailable, as data providers may not wish to publish their server's configuration. Second, they can require significant data transfer, which deteriorates SPARQL query processing performance. Finally, they can only be applied to the class of strongly bounded SPARQL queries to ensure complete and correct evaluation results.

2.2.2 Linked Data Fragments

Linked Data Fragments (LDF) [38, 72] restrict the server interface to a fragment of the SPARQL algebra, to reduce the complexity of queries evaluated by the server and increase its availability. LDF servers are no longer compliant with the W3C SPARQL protocol, and SPARQL query processing is distributed between an LDF server and a smart client. The latter acts as a SPARQL query engine hosted on the client-side. It decomposes the input SPARQL query to evaluate the fragment supported by the server and handles the remaining operations client-side.

Hartig et al. [38] formalized this approach using the Linked Data Fragment machines (LDFM). An LFDM captures possible client-server systems that execute user queries, issued at the client-side, over a server-side RDF dataset. The client communicates with the server using a *server language* $\mathcal{L}_S$ which represents the SPARQL operations supported by the server interface. Then, the client builds final query results from the server responses using a *client response-combination language* $\mathcal{L}_C$, which is an algebra over the server responses. Thus, an LDFM M is a tuple $M = \langle \mathcal{L}_S, \mathcal{L}_C \rangle$. Notice that, if an LDFM wants to allow execution of any SPARQL queries, then it must hold that $\mathcal{L}_S \cup \mathcal{L}_C = \text{CoreSPARQL}$, where CoreSPARQL is the full SPARQL query language [56].

The Triple Pattern Fragments (TPF) approach [72] is one implementation of LDF where the server only evaluates *paginated triple pattern queries*. Formally, a TPF client-server is an LDFM M_{TPF} where $\mathcal{L}_S = \{tp\}$ and $\mathcal{L}_C = \text{CoreSPARQL} \setminus \mathcal{L}_S$. To evaluate Basic Graph Patterns, the TPF client follows an approach similar to the NESTED strategy from [7], illustrated in Figure 2.11. For a BGP $P = \{tp_1, tp_2\}$, the client first evaluates tp_1 using the paginated server interface,

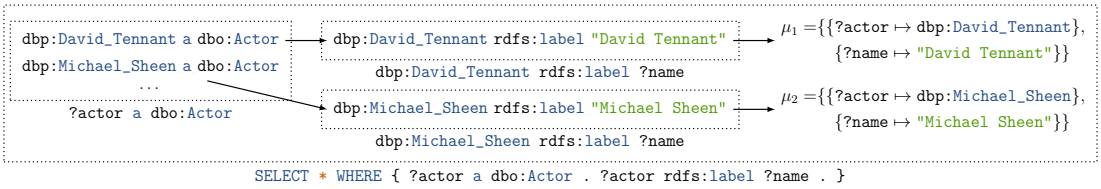


Figure 2.11: Basic Graph Pattern evaluation using the TPF Smart client [72]

retrieves pages of relevant RDF triples, and converts them into solution mappings for tp_1. Then, for each mapping $\mu \in [\![tp_1]\!]_{\mathcal{D}}$, it evaluates the subquery $\mu(tp_2)$ using the server interface, and for each $\mu' \in [\![\mu(tp_2)]\!]_{\mathcal{D}}$, it produces $\mu \cup \mu'$ as a set of join results. For evaluating a Basic Graph Pattern of more than two triple patterns, the TPF client takes advantage of the metadata published by the TPF server to perform a client-side join ordering and always evaluates the most selective triple pattern. As paginated triple pattern queries can be evaluated in bounded time [42], the server does not suffer from the convoy effect. But, as the client computes joins locally, the transfer of intermediate results leads to poor query execution performance. For example, the evaluation of the query Q_1, from Figure 1.3, which finds all actors and their birth places in the DBpedia knowledge graph, using the TPF approach sends 507156 subqueries and transfers 2GB of intermediate results in more than two hours.

Other LDF approaches have introduced different trade-offs, compared to the original TPF approach, to reduce the overhead in data transferred. The Bindings-Restricted Triple Pattern Fragments (BrTPF) approach [35] improves the TPF approach by using the bind-join algorithm [29] to reduce transferred data. While the extended server interface is resilient to convoy effects, the client still executes joins locally. In [2], Acosta et al. propose to use networks of Linked Data Eddies to allow the TPF client to switch between several join strategies. The smart client uses adaptive *query processing techniques* [10, 20] to dynamically adapt query execution to adjust to changes in runtime execution and data availability. In scenarios with unpredictable transfer delays and data distributions, Linked Data Eddies outperform the existing LDF approach. However, the restricted server interface still leads to significant data transfers.

Synthesis While the LDF approaches remain limited by the high data transfer rates, they demonstrate that one can publish RDF data using a less expressive SPARQL query server but still allows the execution of the complete SPARQL query language. As hinted in [38], a more efficient LFDM is a one that will enable for server-side processing of Basic Graph Patterns without generating convoy effects. To the best of our knowledge, none of the existing LDF approaches achieve that.

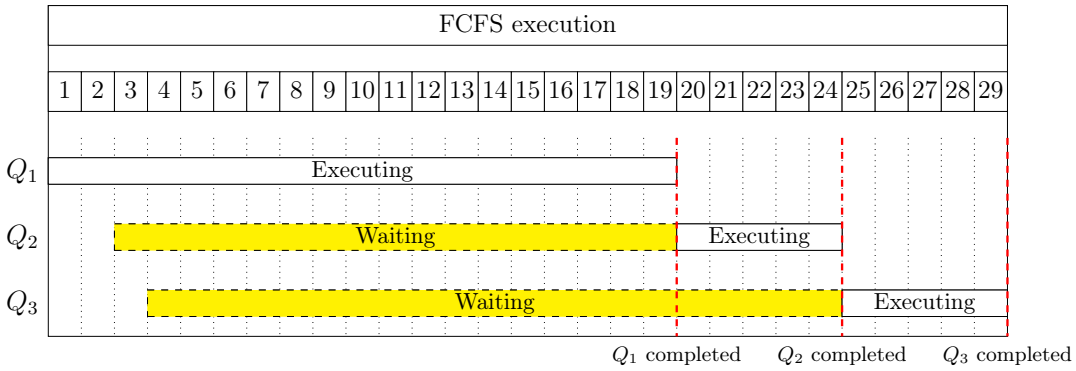


Figure 2.12: Execution of three SPARQL queries using the First-Come First-Served (FCFS) execution policy [25]

2.2.3 Scheduling policies and Preemption

Operating systems have heavily studied First-Come First-Served (FCFS) scheduling policies and the convoy effect [16]. 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 illustrate, consider the execution scenario from Figure 2.12. Three distinct web clients want to execute one SPARQL query each, Q_1, Q_2, and Q_3, using the same SPARQL endpoint. Q_1, Q_2, Q_3 execution times are respectively 20 seconds, 5 seconds, and 5 seconds. We also consider that the server only offers one worker thread, so it can only execute one query at a time. In this scenario, Q_1 arrives first at the server at t_1 and runs for 20 seconds. At t_3, the server receives Q_2, but has to wait for execution has the only worker is still busy with Q_1. Similarly, Q_3 is received at t_4 and must wait. At t_{20}, Q_1 execution completes, and the first client receives its results. Then, the server starts the execution of Q_2, while Q_3 is still in the waiting queue. At t_{25}, Q_2 execution completes, and the server begins the evaluation of Q_3, which finishes at t_{30}. In the end, the first web client has received its query results after 20 seconds, *i.e.*, the execution time of Q_1, while the two other clients have received their results after 25 seconds and 30 seconds, respectively. Hence, the execution of Q_2 and Q_3 have been delayed by the evaluation of Q_1 because the FCFS execution policy created a convoy effect. In terms of query execution performance, we have an average waiting time of $\frac{0+17+22}{3} = 13$ seconds, an average query completion time of $\frac{20+25+30}{3} = 25$ seconds and an average time for first results of also 25 seconds.

To avoid convoy effects while preserving scheduling performance, researchers designed various families of scheduling algorithms, also called *scheduling policies*. In particular, the *Round-Robin* (RR) scheduling algorithm [46] provides a fair

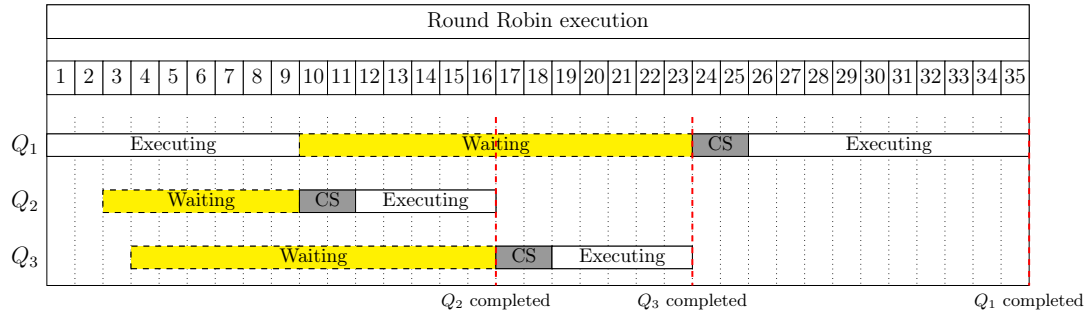


Figure 2.13: Execution of the three SPARQL queries from Figure 2.12 using the Round Robin scheduling policy.

allocation of CPU between tasks, avoids 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. This action of suspending a task to resume it later is called *preemption*. Figure 2.13 shows the execution scenario from Figure 2.12 executed under a RR scheduling policy. Jobs run for a time quantum of 10 seconds, after which the scheduler triggers a *context switch* (CS) operation to suspend the current task and resume the next one. In our example, context switching takes about two seconds. In the RR algorithm, the value of this time quantum is critical for its performance. If it is too high, RR behaves like FCFS with the same issues, and when it is too low, the overhead of context switching dominates the overall performance.

Scheduling algorithms to organise query execution at SPARQL endpoints [49, 70] has been considered to reduce the impact of convoy effects. However, all of these approaches require to know the *cost* of incoming SPARQL queries, *e.g.*, the query execution time or the CPU and memory used. For a public SPARQL query server, this cost is hard to estimate as it can accept any type of query [40]. Furthermore, these approaches only consider the scheduling issue at the scope of operating systems, *i.e.*, **only between running tasks, excluding those waiting in the server's queue**. So, while the operating systems may use the RR algorithm, or any equivalent scheduling policy, the whole SPARQL query execution model still follows the FCFS policy. If we want to build a *fully preemptive Web server*, we need to "zoom-out" and **consider the queries sent to the server as the tasks and the Web server as the resource that tasks competed to get access**. Currently, no SPARQL query server provides such preemption.

Chapter 3

Web preemption for public SPARQL Query Servers

As stated before, quotas are an essential part of the Web ecosystem, as they allow data providers to regulate how users consume their services. In this thesis, we hypothesize that the issue with time quotas is not that they interrupt queries, but the impossibility for the client to *resume* the query execution afterward. Inspired by the work on time-sharing scheduling policies, we propose a new query execution model for SPARQL query servers, called *Web Preemption*.

We define Web preemption as the capacity of a Web server to suspend a running query after a fixed quantum of time and resume the next waiting query. When the server suspends a query, it returns the query's partial results and suspended state S_i to the Web client, which can then resume query execution by sending S_i back to the Web server. Compared to an FCFS scheduling policy, our results show that Web preemption provides *a fair allocation of Web server resources across queries, a better average query completion time per query, and a better time for first results* [5].

3.1 The Web preemption execution model

3.1.1 Model definition

We now formally define the Web preemption execution model. We consider a *preemptive Web server*, which hosts a *read-only* RDF dataset, and a *smart Web client*, that evaluates SPARQL queries using the server. The server has a pool of *server workers* parameterized with a fixed time quantum. Workers are in charge of query execution. The server also has a *waiting queue* to store incoming queries when all workers are busy. We consider an *infinite* population of clients, a *finite*

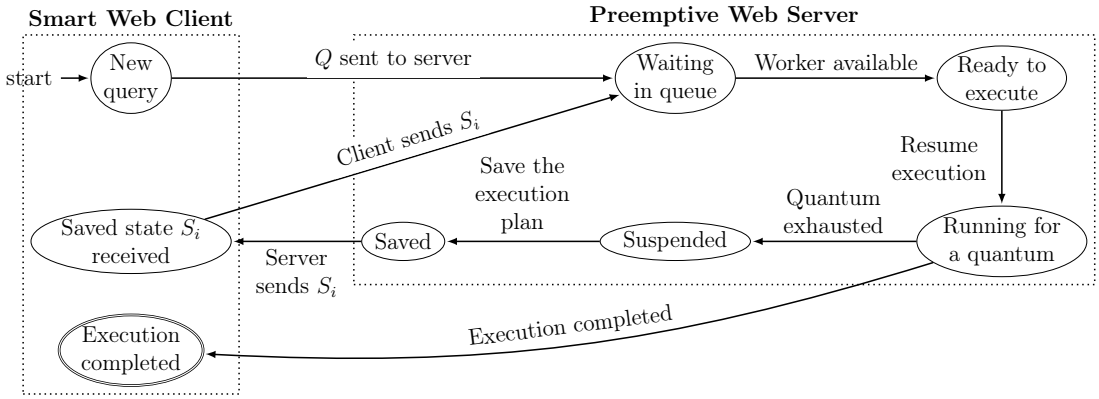


Figure 3.1: Possible states of query execution in a preemptive Web Server.

server queue, and a *finite* number of server workers.

A preemptive Web server supports three operations: `Execute`, `Resume`, and `Suspend`. The `Execute` operation is used by the server to execute a query Q for a time quantum q. It takes as input a running query Q_i represented by its *physical query execution plan*, denoted P_{Q_i}, and executes it for the duration q. Then, it produces a *set of solution mappings* ω_i and the execution plan P'_{Q_i}, which is the state of P_{Q_i} after the execution ($\texttt{Execute}(P_{Q_i}) = \langle \omega_i, P'_{Q_i} \rangle$). The `Suspend` operation, which saves the query execution state, is applied to P_{Q_i} and produces a *saved state* S_i ($\texttt{Suspend}(P_{Q_i}) = S_i$). The `Resume` operation, which resumes query execution from a saved state, is the inverse operation: it takes S_i as a parameter and restores the physical query execution plan ($\texttt{Resume}(S_i) = P_{Q_i}$).

Figure 3.1 presents possible states of a query. The transitions are executed either by the Web server or by the client. The Web server accepts, in its waiting queue, Web requests containing either SPARQL queries or suspended queries. If a worker is available, it picks a query from the waiting queue. For a query Q_i, the worker produces a physical query execution plan P_{Q_i} using the optimize-then-execute paradigm [27] and starts its execution for a time quantum. For a saved state S_i, the server resumes the execution of the corresponding query Q_i using the `Resume` operation. The time to resume the physical query execution plan for a query is not deducted from the quantum.

If a query completes before the time quantum, then the server returns the results to the Web client, and the query execution is complete. If the time quantum is exhausted and the query is still running, then the server uses the `Suspend` operation to suspend execution and produce a saved state S_i. The time to suspend and save the query is not deducted from the quantum. Next, the server returns a *page of results* p_i to the client, which is a tuple $p_i = \langle \omega_i, S_i \rangle$ where ω_i is the set of solution mappings produced during the time quantum q_i, and S_i is the saved plan obtained

when suspending the running query at the end of q_i. The Web client is then free to continue the query execution by sending S_i back to the Web server.

Consequently, the evaluation of a SPARQL query using a preemptive Web server create a *partition of solution mappings over time*, where the Smart client retrieves partial set of solution mappings $\omega_1, \ldots, \omega_n$ over the duration of query execution. Given a RDF Dataset $\mathcal{D}$ and a preemptive Web Server with time quantum q. The correct evaluation of a SPARQL query Q over $\mathcal{D}$ using the server produces a finite set of pages $\mathcal{P} = \{p_1, \ldots, p_n\}$ which have the following properties:

1. $[\![Q]\!]_{\mathcal{D}} = \bigcup_{\langle \omega_i, S_i \rangle \in \mathcal{P}} \omega_i$.
2. It holds that, for $p_n = \langle \omega_n, S_n \rangle, S_n = nil$.
3. For every pair of two distinct pages $p_i = \langle \omega_i, S_i \rangle \in \mathcal{P}, p_j = \langle \omega_j, S_j \rangle \in \mathcal{P}$, it holds that $S_i \neq S_j$.
4. There exists a strict total order $\prec$ on $\mathcal{P}$ such as, $\forall p_i = \langle \omega_i, S_i \rangle \in \mathcal{P}, \exists p_j = \langle \omega_j, S_j \rangle \in \mathcal{P}$, where p_j is the direct successor of p_i according to $\prec$, it holds that $\texttt{Execute}(\texttt{Resume}(S_i)) = \langle \omega_j, x \rangle$ and $\texttt{Suspend}(x) = S_j$.

Property 1 states that the client receives all query solutions ($[\![Q]\!]_{\mathcal{D}}$) after the last page is produced. Property 2 asserts that the last page produced does not contains any saved state: query execution has completed and the server cannot resume it past this point. Properties 3 and 4 states that *query execution progresses over time*: all saved states are distinct and produced linearly in time.

3.1.2 Illustration

To illustrate our model, consider the execution scenario from Figure 3.2, which extends the one from Figure 2.12. Recall that we have three distinct web clients who want to execute one SPARQL query each, Q_1, Q_2, and Q_3, and their execution times are respectively 20 seconds, 5 seconds, and 5 seconds. We also consider that the server only offers one worker thread, so it can only execute one query at a time. In this scenario, Q_1 arrives first at the server, then Q_2 and Q_3 arrive at t_3 and t_4, respectively. We have seen in Chapter 1 that, if the server follows a First-Come First-Served scheduling policy, we have an average waiting time of $\frac{0+17+22}{3} = 13$ seconds, an average query completion time of $\frac{20+25+30}{3} = 25$ seconds and an average time for the first results of also 25 seconds.

Now, we review the same execution scenario, but under the Web Preemption execution model, with a time quantum of 10 seconds. In this case, the server *suspends* the execution of Q_1 after 10 seconds, saves the execution state, and returns it to the client. Then, the server executes Q_2 and Q_3 in sequence, and both queries complete under the time quantum. At t_{19}, the server has received the saved state of Q_1, but Q_3 is still running, so the server put the saved state in the waiting queue. At t_{23}, the server *resumes* the execution of Q_1 from its saved state, and the query runs for the remaining 10 seconds. For this scenario, we have an

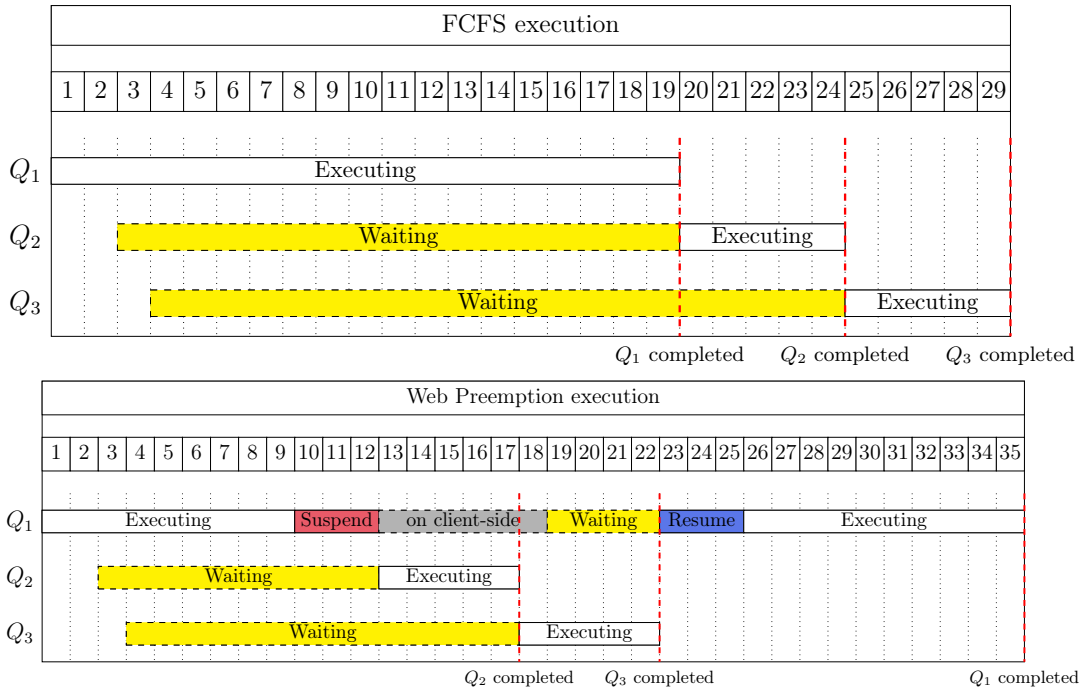


Figure 3.2: First-Come First-Served (FCFS) policy compared to Web Preemption (time quantum of 10s and overhead of 3s).

average waiting time of $\frac{4+10+14}{3} = 9.3$ seconds, an average query completion time of $\frac{35+16+19}{3} = 23.3$ seconds, and an average time for the first results of $\frac{12+16+19}{3} = 15.6$ seconds. So, compared to the FCFS scenario, Web Preemption reduces the average waiting time of 4 seconds ($\approx 30\%$), the average query completion time of 2 seconds, and the average time for the first results of 10 seconds ($\approx 40\%$).

Notice that, in this scenario, suspending and resuming a query takes about 3 seconds. This duration is called the *overhead* of Web Preemption and significantly impacts query execution performance. Consequently, the main challenges with Web preemption are *to bound the preemption overhead in time and space* and *determine the time quantum* required to amortize the overhead.

3.2 Challenges of implementing the Web preemption model

The Web Preemption execution model presents several challenges when it comes to its implementation, which all relate to the minimization of the preemption overhead. We now review these challenges and will propose their solutions in the next chapter.

3.2.1 Minimizing the preemption overhead

As said before, the main bottleneck of the Web Preemption model is its overhead. So, we need to bound the time and space complexities of the `Suspend` and `Resume` operations to the smallest complexity possible. Informally, these complexities correspond to the following:

- The time complexity of `Suspend` is the complexity of interrupting and then saving a physical query execution plan.
- The time complexity of `Resume` is the complexity of resuming a physical query execution plan from a saved state.
- The space complexity of `Suspend` is the size of the saved state produced after saving a physical query execution plan.

Ideally, we want these complexities to only depends on the query evaluated, and not on the size of the RDF dataset. Otherwise, the preemption overhead will grow with the size of the data, which is intractable in the context of the Web of Data, where the data volume is always increasing [58, 60]. So, we want our complexities to near a complexity of $O(|Q_i|)$, where $|Q_i|$ is the number of operators in the physical query execution plan used to execute Q_i. It requires to suspend, save, or resume the state of all physical query operators in any possible physical query execution plan in near-constant time, *i.e.*, $O(1)$.

However, *some physical SPARQL query operators need to materialize data in their internal states*, *i.e.*, build collections of mappings collected from other operators to perform their actions. To illustrate, recall the Sort iterator from Table 2.1, used to evaluate the `ORDER BY` SPARQL operator. This physical operator first needs to store all input solution mappings in a table before producing any results. If the SPARQL query has n solution mappings, then the time complexity of suspending and saving the internal state of the Sort iterator is in $O(n)$. Additionally, if we consider a stateful Web Preemption, we would need to ship these n results between the client and the server each time preemption occurs. This situation could generate significant data transfer and deteriorate query execution performance. For resuming physical operators, similar reasoning holds.

Consequently, we distinguish two categories of physical SPARQL query operators: *mapping-at-a-time* and *full-mapping* operators, formalized in Definition 4. Database systems made a similar distinction between *tuple-at-a-time* and *full-relation* operators [26] (see Chapter 15.2).

Definition 4 (Types of physical query operators)**.** *A physical SPARQL query operator* OP *is a* ***mapping-at-a-time*** *operator if, at any time, it stores at most one*

set of solution mappings in its internal state. Otherwise, OP *is a* ***full-mappings*** *operator.*

From Definition 4, we have the intuition that full-mapping operators are going to be more expensive to suspend and resume than mapping-at-a-time operators. Thus, the first challenge of implementing the Web preemption model is to suspend and resume any kind of operators efficiently, to achieve the lowest bound on time and space complexities.

3.2.2 Choosing between stateless and stateful preemption

In the Web Preemption model, there are two possible modes for producing and storing saved states: *stateless* or *stateful*. In stateless mode, the server directly returns saved states to the client, alongside query results. This strategy, widely used for building public Web APIs, provides several properties:

- **Fault-tolerance**: As saved states are saved client-side, any client can resume query execution even if the server has restarted due to a crash.
- **Scaling**: The data provider can replicate the server to scale dynamically under the load without having to copy saved states across replicas.
- **Load balancing**: All preemptive Web servers that host the same RDF dataset can resume the same saved state. Thus, the data provider can balance the load of query processing across several replicated servers.

However, the stateless mode exhibits two main flaws. First, the server must send all saved states through the HTTP protocol, so the preemptive Web server needs to encode them in a compatible format. This *serialization phase* increases the time complexity of the `Suspend` and `Resume` operations, as more compressed formats are more expensive to encode and decode. Second, the size of saved states increases the data transfers between the client and the server. So, the preemptive Web server works under even stringent constraints on the space complexity of the `Suspend` operation, which further restricts the space of physical query operators available for SPARQL query processing. As an example, the Sort iterator, from Figure 2.1, can increase a lot the preemption overhead in stateless mode, as it needs to store all query results in its internal state. Hence, it might be excluded by the SPARQL query engine when evaluating an ORDER BY operator.

At the opposite, in stateful mode, all saved states are stored server-side, and the clients receive only references. For example, saved states can be assigned unique identifiers and stored in a fast-access in-memory database. The clients then only receive these identifiers as saved states, and when they sends them

back to the server, it accesses the in-memory database to get the associated saved plans. Stateful mode allows the preemptive Web server to use cheaper serialization strategy for saved states, as they never need to be transmitted over the network. It effectively removes the increased preemption overhead that occurs in stateless mode, and increase the space of physical query operators available to the server. However, in stateful mode, we lose the fault-tolerance, scaling, and load balancing properties compared to the stateless mode. As saved states are stored on the server-side, replication becomes more complex. We need to either replicate saved states or store them in a centralized location. In the case of server crashes, saved states can be lost, which forces the clients to restart their queries *from the beginning*.

Stateless and stateful modes present different tradeoffs when it comes to saving query execution plans and saving their saved states. So, an implementation of the Web Preemption model must choose between stateless preemption, and favor fault-tolerance and scaling over query processing performance, or a stateful preemption, and choose the opposite tradeoff.

3.2.3 Communication between physical query operators

When the server saves the sate of a physical query execution plan, we also need to save the intermediate results exchanged between operators in the plan. So, the implementation chosen by the SPARQL query engine for communication inside the plan has a direct impact on the space complexity of the `Suspend` operation. If the query engine wants to use temporary files to store intermediate results, then we also need to save these files when preemption occurs, and in the case of stateless web preemption, we need to *send them to the client*.

Communication between operators has no direct relationship with the use of full mappings operators, as a plan entirely composed of mapping-at-a-time operators can create significant inter-operator communication. For example, under the Vectorized processing model [78], mapping-at-a-time operators can exchange large vectors of intermediate results while only saving a single set of solution mappings in their internal state. Thus, an implementation of the Web Preemption model has to care with how it handles inter-operator communication, as it impacts the preemption overhead.

3.2.4 Saving consistent states of the physical query execution plan

When suspending a physical query execution plan, the SPARQL query engine has to visit each operator in the plan and interrupt it *a consistent state* from which it can be resumed later. However, physical query operators may contain *non-interruptible*

sections: query execution cannot be interrupted until exiting those sections. This concept is similar to the notion of *kernel sections* in operating systems [5], which are sections of code that cannot be interrupted by the scheduler.

Non-interruptible sections can occur in any physical query operator. For example, an operator that reads data from the disk will want to consider the retrieval of each data item as a non-interruptible operation. Otherwise, the operator could be interrupted while reading bytes from the disk, which can result in a corrupted state or cancel the operation indefinitely. In a sense, *non-interruptible sections help to provide progression to the query execution*, as they define atomic units of progress in physical query operator.

However, the usage of these sections can have an unexpected side-effect on the preemption overhead. Indeed, when preemption occurs, if a physical query operator is in such a section, then the query engine must wait for the operator to exit the section before interrupting it. Thus, the time complexity of the non-interruptible sections impacts the time complexity of the `Suspend` operation: the longer these sections are, the longer it will take to suspend the running query. In the worst case, they can re-introduce convoy effects. In [16], Blasgen et al. observe this effect when the usage of locks in concurrency control adds unwanted waiting time in query processing. Thus, an implementation of Web Preemption must minimize the duration of non-interruptible sections, at the risk of increasing the preemption overhead.

Chapter 4

SaGe: A preemptive SPARQL query engine

The Web preemption model allows data providers to build SPARQL query servers that are not subject to convoy effects and always deliver complete results for any SPARQL queries. However, as seen at the end of the previous chapter, we are left with a significant issue related to the implementation of this model: Given a SPARQL query Q, we need to minimize the time and space complexities of the `Suspend` and `Resume` operations when applied to Q.

In this chapter, we propose SaGe, a SPARQL query engine that implements the Web Preemption model and solves the issue above. Our objective is to bound these complexities such that they depend only on the number of operations in the query plan. Consequently, *the problem is to determine which physical query plans P_Q has a preemption overhead bounded in $\mathcal{O}(|Q|)$*, where $|Q|$ denotes the number of operators in the expression tree of P_Q.

4.1 Implementing the Web Preemption model

In the previous chapter, we have observed that the preemption overhead for full mappings physical operators is likely to exceed our target bound of $\mathcal{O}(|Q|)$. To overcome this issue, we propose and motivate a serie of design choices for implementing the Web Preemption model, in order to minimize the preemption overhead. Bear in mind that we propose here *one feasible implementation* of the Web Preemption model, but no the only one.

First, to reduce the preemption overhead of full-mappings operators, we choose to distribute operators between a SaGe server and a SaGe Smart Web client as follows.

- *Mapping-at-a-time* operators are suitable for Web preemption, so they are

```
PREFIX schema: <http://schema.org/contentSize/>
PREFIX gr: <http://purl.org/goodrelations/>
SELECT DISTINCT ?vo ?v1 WHERE {
 ?v0 gr:includes ?v1. # tp1
 ?v1 schema:contentSize ?v3. # tp2
}
```

Figure 4.1: SPARQL query Q_3, from the WatDiv benchmark [4].

supported by the SaGe server. They are a subset of CoreSPARQL [38], composed of Triple Patterns, Basic Graph Patterns, `UNION`, `FILTER`, and `SELECT` operations. We explain how to implement them in Section 4.2.

- *Full-mappings* operators are not suitable for Web preemption, so they are implemented in the SaGe Smart Web client. They are physical operators used to implement `OPTIONAL`, `SERVICE`, `ORDER BY`, `GROUP BY`, `DISTINCT`, `MINUS`, `FILTER EXIST`, and aggregations operations(`COUNT`, `AVG`, `SUM`, `MIN`, `MAX`). We explain how to implement them in Section 4.3.

As proposed in LDF [38, 72], the collaboration of the SaGe client and the SaGe server allows to execute full SPARQL queries. Hence, SaGe is an LDFM M_{SaGe} where $\mathcal{L}_S \cup \mathcal{L}_C = \text{CoreSPARQL}$.

Regarding the communication between operators, we choose to follow the iterator model [27, 28] to avoid the materialization of intermediate results between physical operators. To support preemption, we extend standard iterators to *preemptable iterators*, formalized in Definition 5. As we connect iterators in a pipeline, we consider that each iterator is also responsible for recursively stopping, saving and resuming its predecessors.

Definition 5 (Preemptable iterator)**.** *A* preemptable iterator *is an iterator that supports, in addition to the classic* `Open`*,* `GetNext`*, and* `Close` *methods [28], the following methods:*

- `Stop` *waits for all non-interruptible sections to complete, and then it interrupts the iterator and its predecessor(s).*
- `Save` *serializes the current state of the iterator and its predecessor(s) to produce a saved state.*
- `Load` *reloads the iterator and its predecessor(s) from a saved state.*

Finally, we choose to implement the SaGe preemptive SPARQL query server under the stateless mode and return the saved pipeline of iterators to the client.

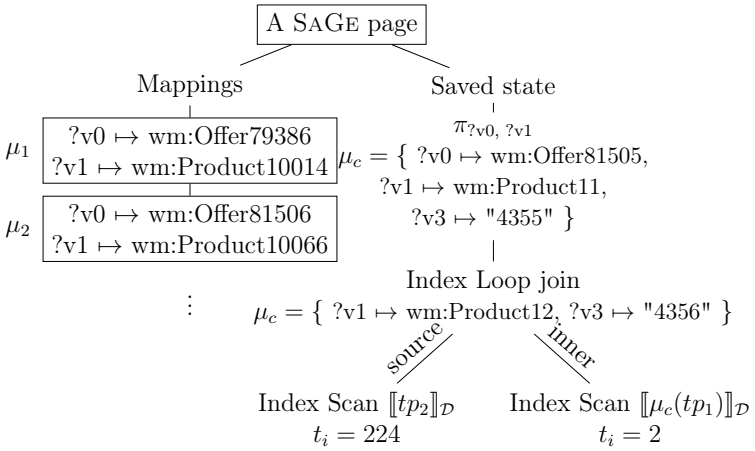


Figure 4.2: A tree representation of a page returned by the SaGe server when executing SPARQL query Q_3 with the saved plan passed by value.

Algorithm 1: Implementation of the `Suspend` and `Resume` functions following the iterator model

Require: $\mathcal{I}$: pipeline of iterators, S: serialized pipeline state (as generated by `Suspend`)

```
1 Function Suspend(I):
2     let root←first iterator in I
3     Call root.Stop()
4     return root.Save()
5 Function Resume(I, S):
6     let root←first iterator in I
7     Call root.Load(S)
8     return I
```

To illustrate, consider Figure 4.2, which shows a results page as returned by the SaGe server when executing the SPARQL query Q_3 from Figure 4.1. Thus, we favour a fault-tolerant implementation of Web preemption. The state of the Scan operator contains the *id* of the last triple read t_i, and the state of the Index Loop Join operator contains mappings pulled from the previous operator and the state of the inner scan of tp_1.

4.2 The SaGe preemptive SPARQL Query Server

The SaGe server supports the evaluation of `SELECT`, Triple Patterns, Basic Graph Patterns, `UNION`, and `FILTER` operations. The logical and physical query plans are built following the *optimize-then-execute* [27] paradigm.

Algorithm 1 presents the implementation of the `Suspend` and `Resume` operations for a pipeline of preemptable query iterators. `Suspend` simply stops and saves

Preemptable iterator	**Space complexity of** `Suspend`	**Time complexity of**	
		`Suspend`	`Resume`
Projection $\pi_V(P)$	$\mathcal{O}(\lvert V\rvert + \lvert\, dom(P)\rvert)$ Proposition 1	$\mathcal{O}(\lvert V\rvert + S_P)$ Proposition 2	$\mathcal{O}(R_P)$ Proposition 3
Index Scan $IScan(tp)$	$\mathcal{O}(\lvert tp\rvert + \lvert t\rvert)$ Proposition 4	$\mathcal{O}(\log_b \lvert\mathcal{D}\rvert)$ Proposition 5	$\mathcal{O}(\log_b \lvert\mathcal{D}\rvert)$ Proposition 6
Merge Join $MJoin(P_1, P_2)$	$\mathcal{O}(\lvert\, dom(P_1)\rvert + \lvert\, dom(P_2)\rvert)$ Proposition 7	$\mathcal{O}(S_{P_1} + S_{P_2})$ Proposition 8	$\mathcal{O}(R_{P_1} + R_{P_2})$ Proposition 9
Index Loop Join $IJoin(P, tp)$	$\mathcal{O}(\lvert\, dom(P)\rvert + \lvert tp\rvert + \lvert t\rvert)$ Proposition 10	$\mathcal{O}(S_P + \log_b \lvert\mathcal{D}\rvert)$ Proposition 11	$\mathcal{O}(R_P + \log_b \lvert\mathcal{D}\rvert)$ Proposition 12
Multi-set Union $MUnion(P_1, P_2)$	$\mathcal{O}(\lvert\, dom(P_1)\rvert + \lvert\, dom(P_2)\rvert)$ Proposition 13	$\mathcal{O}(S_{P_1} + S_{P_2})$ Proposition 14	$\mathcal{O}(R_{P_1} + R_{P_2})$ Proposition 15
Filter $Filter(P, \mathcal{R})$	$\mathcal{O}(\lvert\, dom(P)\rvert + \lvert\mathcal{R}\rvert)$ Proposition 16	$\mathcal{O}(S_P)$ Proposition 17	$\mathcal{O}(R_P)$ Proposition 18
Physical plan P_Q	$\mathcal{O}(\lvert\, dom(Q)\rvert + \lvert Q\rvert \times \lvert t\rvert)$ Theorem 1	$\mathcal{O}(\lvert Q\rvert \times \log_b \lvert\mathcal{D}\rvert)$ Theorem 2	$\mathcal{O}(\lvert Q\rvert \times \log_b \lvert\mathcal{D}\rvert)$ Theorem 3

Table 4.1: Complexities of preemption for preemptable iterators.

each iterator recursively in the pipeline, and `Resume` reloads the pipeline in the suspended state using a saved state. To illustrate, consider the SaGe page shown previously in Figure 4.2. This page contains the plan that evaluates the SPARQL query Q_2 with $|Q_2| = 4$ operators. A preemptable projection operator implements the `SELECT` operator, a preemptable Index Scan implements the evaluation of the evaluation tp_2, and a preemptable Index loop join implements the join between tp_2 and tp_1. In this saved state, the evaluation of tp_2 has been suspended after scanning 224 relevant triples, while the Index Loop join with tp_1 has been suspended after scanning two triples from the inner loop.

In the following, we review SPARQL operators implemented by the SaGe server as preemptable iterators. We modify the regular implementations of these operators to include a non-interruptible section when needed. Operations left unchanged are not detailed. Notably, this includes the `Stop` operation, as its algorithm is the same for every preemptable iterator: it waits for all non-interruptible sections to complete and then suspends the operator. Table 4.1 resumes the complexities related to the preemption of server operators, where S_P and R_P denote the time complexity for suspending and resuming the iterator P, respectively, $|Q|$ denotes the number of operators in a plan, and $|t|$ and $|tp|$ denote the size of encoding an RDF triple and a triple pattern, respectively.

4.2.1 SPARQL query forms

The SPARQL language defines four query forms [1], which use the set of solutions mappings produced by the evaluation of the `WHERE` clause of the query to query results of various types. These query forms are:

- **SELECT** Returns a subset of the variables bound in a query group pattern [56]. It is similar to the *projection* operator in relational databases [26].
- **CONSTRUCT** Returns an RDF graph constructed by substituting variables in a set of triple templates [47].
- **ASK** Returns a boolean indicating whether a query pattern matches or not, *i.e.*, if the evaluation of the query produces *at least* one result.
- **DESCRIBE** Returns an RDF graph that describes the resources found.

Algorithm 2: A *Preemptable Projection Iterator* $\pi_V(P)$

Require: V: set of projection variables, I: predecessor in the pipeline
Data: μ: set of solution mappings

```
1  Function Open():
2      I.Open()
3      μ ← nil
4  Function GetNext():
5      if μ = nil then
6          μ ← I.GetNext()
7      non interruptible
8          let μ' ← Projection(μ, V)
9          return μ'
10 Function Close():
11     I.Close()
12 Function Save():
13     s ← I.Save()
14     return ⟨s, μ⟩
15 Function Load(s', μ'):
16     I.Load(s')
17     μ ← μ'
```

The SaGe query server evaluates SELECT queries using the *preemptable projection operator* $\pi_V(P)$ [62]. It performs the projection of mappings obtained from its predecessor P according to a set of projection variables $V = \{v_1, \ldots, v_k\}$. Algorithm 2 gives the implementation of this iterator. In this algorithm, we need to ensure that, when preemption occurs, the operator does not discard mappings without applying projection to them. When the `GetNext()` is called, the operator checks if the local variable μ has a value. If not, it pulls a set of solution mappings from its predecessor in the pipeline and stores it in μ Then, at Lines 7-9, it uses a

[1] https://www.w3.org/TR/rdf-sparql-query/#QueryForms

non-interruptible section to save the projection results in a temporary variable μ', then it discards the last μ read from the pipeline, and returns μ' as a new query result. Thus, when preemption occurs during the execution of `GetNext()`, only two cases can arise: 1) μ has no value, so the iterator will restart by pulling a new value from the pipeline, and 2) μ has a value, so the iterator resume the projection operation.

The `Save()` and `Load()` method are simple for this operator: the first saves the current value of μ, and the latter reloads this value from the saved state. Propositions 1, 2, and 3 give the preemption overhead for the preemptable projection iterator. As the proofs of these propositions use the same structure as the other proofs in this chapter, we write them in detail, and we will simplify for the others.

Proposition 1. *The space complexity of suspending a preemptable projection iterator $\pi_V(P)$ is in $\mathcal{O}(|V| + |\operatorname{dom}(P)|)$, where $\operatorname{dom}(P)$ is the set of variables bound in solution mappings produced by the predecessor iterator P.*

Proof. The `Save()` function in Algorithm 2 saves the set of variables V and the current value of the local variable μ, which can either be *nil* or a set of solution mappings read from the predecessor iterator P. The space complexity of a set of solution mappings μ is in $\mathcal{O}(|dom\ (\mu)|)$. So if we consider $dom(P)$ as the set of variables bound in solution mappings produced P, the space complexity of the `Suspend()` operation in Algorithm 2 is in $\mathcal{O}(|V| + |\ dom(P)|)$. □

Proposition 2. *The time complexity of the `Suspend` operation applied on a preemptable projection iterator $\pi_V(P)$ is in $\mathcal{O}(|V|+S_P)$, where S_P is the time complexity for suspending the iterator P.*

Proof. To compute the time complexity of the `Suspend` applied to a preemptable iterator, we need to compute bounds on: 1) the duration of the iterator's non-interruptible sections, 2) the time complexity of the iterator's `Save` function.

In Algorithm 2, there is only one non-interruptible section from Line 7 to 9. It contains three operations: a projection operation, an assignment, and a function return call. The first one selects a subset of the variables bound in a set of solution mappings ($\mathcal{O}(|V|)$), while the other one performs an assignment to a variable ($\prime(1)$). So, the complexity of the projection predominates, and the non-interruptible section from Line 7 to 9 is bounded in $\mathcal{O}(|V|)$.

Next, for the `Save` function, its time complexity is dominated by the operation of saving the predecessor iterator P. So, if S_P is the time complexity for suspending the iterator P, the time complexity of `Save` is bounded in $\mathcal{O}(S_P)$.

Consequently, the time complexity of suspending and saving a preemptable projection iterator $\pi_V(P)$ is in $\mathcal{O}(|V| + S_P)$. □

Algorithm 3: A *Preemptable Index Scan Iterator*, evaluating a triple pattern tp using a clustered index over the RDF dataset

Require: tp: triple pattern, $\mathcal{D}$: RDF dataset, $\mathcal{I}_{tp}$: clustered index over tp
Data: t: last matching RDF triple read

```
1  Function GetNext():
2      non interruptible
3          t ← next RDF triple matching tp in D
4          let μ ← set of solutions mappings such as μ(t) = tp
5          return μ
6  Function Save():
7      return ⟨tp, t⟩
8  Function Load(tp', t'):
9      tp ← tp'
10     t ← I_tp.Locate(t')          // Locate the last triple read
```

Proposition 3. *The time complexity of resuming a preemptable projection iterator $\pi_V(P)$ is in $\mathcal{O}(R_P)$, where R_P is the time complexity for resuming the iterator P.*

Proof. The operation of resuming the predecessor iterator P dominates the time complexity of the `Load()` function in Algorithm 2. So, if R_P is the time complexity for resuming the iterator P, then the time complexity of resuming the preemptable projection iterator $\pi_V(P)$ is in $\mathcal{O}(R_P)$. □

For evaluating CONSTRUCT, ASK, and DESCRIBE queries, we do not need dedicated physical query operators. Indeed, the SaGe Smart Web client can execute them using a single SPARQL SELECT query, followed by some lightweight client-side computation. Given a CONSTRUCT query $Q_c =$ `CONSTRUCT` H `WHERE` P, where H is a set of triple patterns and P is a graph pattern, the answers of Q_c are $[\![Q_c]\!]_{\mathcal{D}} = \{\mu(tp) \mid \forall \mu \in [\![P]\!]_{\mathcal{D}}, \forall tp \in H, \mu(tp)$ is well-formed$\}$ [47]. Thus, to evaluate Q_c, the Smart Web client needs to send a SPARQL query $Q'_c =$ `SELECT * WHERE` P to the server, and then computes the answers of Q_c locally from those of Q'_c. Evaluating an ASK query requires to find at least one solution mappings that satisfy the query group pattern. So, the client can simply send an equivalent SELECT query with a modifier `LIMIT 1`, and output *True* if it receives results. For DESCRIBE queries, the Smart Web client can rewrite them into equivalent CONSTRUCT queries and evaluate the rewritten queries as described before.

4.2.2 Triple Pattern evaluation

The evaluation of a triple pattern *tp* requires sequentially reading an RDF dataset $\mathcal{D}$, locate all matching RDF triples, and produce the corresponding set of solutions mappings. Suspending and resuming this process can be done by saving the *last matching RDF triple read before preemption occurs* and *resume scanning from this triple*, respectively. To do this, we choose to implement the SaGe server as an RDF triple store and rely on clustered indexes to locate RDF triples efficiently using *index scans*. For clustered indexes, we rely on B+-trees [17], as they offer excellent performance for index scans. We use three indexes SPO, OSP, and POS, to achieve optimal coverage for index scans with minimal space overhead [34].

Algorithm 3 gives the implementation of a preemptable index scan for evaluating a triple pattern. The `GetNext()` function scans through RDF triples matching the triple pattern using the clustered index. At each call, it locates the next matching triple, and produces solutions mappings from it. The whole process is a non-interruptible section, as it represents the smallest unit of progression for this iterator. The `Save()` function stores the triple pattern *tp* and the last RDF triple t read. Using the indexes, the `Load()` function locates the last matching RDF triple read t. Propositions 5, 4, and 6 give the preemption overhead for the preemptable Index Scan iterator.

Proposition 4. *The space complexity of suspending a preemptable index scan iterator $IScan(tp)$ is in $\mathcal{O}(|tp| + |t|)$, where $|tp|$ and $|t|$ denote the size of encoding a triple pattern and a RDF triple, respectively.*

Proof. The `Save()` function of Algorithm 3 saves the triple pattern evaluated, and the last RDF triple read before preemption occurs. Thus, its space complexity is in $\mathcal{O}(|tp| + |t|)$. □

Proposition 5. *The time complexity of suspending a preemptable index scan iterator $IScan(tp)$ is in $\mathcal{O}(log_b(|\mathcal{D}|))$, where b is the B+-tree order and $\mathcal{D}$ is the RDF dataset queried.*

Proof. The only non-interruptible section of Algorithm 3 is from Line 2 to 5. The operation that predominates the time complexity of this section is the production of the next matching RDF triple. Since we use three clustered indexes in our triple store, we always scan matching RDF triples as continuous sequences. Thus, when preemption occurs, we can distinguish two cases: 1) The iterator is locating the first matching RDF triple, and 2) The iterator is reading the next matching RDF triple. In the first case, the iterator uses a lookup in the appropriate B+-tree, which is in $\mathcal{O}(log_b(|\mathcal{D}|))$ [17]. In the second case, the iterator jumps to the adjacent triple in the index, which is in $\mathcal{O}(1)$. So, in the worst case, preemption occurs when

the iterator is locating the first matches, so the time complexity of suspending a call to `GetNext()` in Algorithm 3 is in $\mathcal{O}(log_b(|\mathcal{D}|))$. □

Proposition 6. *The time complexity of resuming a preemptable index scan iterator $IScan(tp)$ is in $\mathcal{O}(log_b(|\mathcal{D}|))$, where b is the B+-tree order and $\mathcal{D}$ is the RDF dataset queried.*

Proof. The `Load()` function of Algorithm 3 resumes triple pattern evaluation by locating the last RDF triple read before preemption occurs. As seen in the proof of Proposition 5, such lookup is in $\mathcal{O}(log_b(|\mathcal{D}|))$. □

From Proposition 5 and 6, we observe that the time complexity of suspending and resuming a preemptable index scan iterator depends on a logarithm of the size of the RDF dataset queried. As triple pattern evaluation is the basic building block of any physical SPARQL query execution plan, we conclude that **we cannot meet our bound of $\mathcal{O}(|Q|)$ on the preemption overhead**. Consequently, we propose a new target lower bound, in $\mathcal{O}(|Q| \times log_b(|\mathcal{D}|))$. It means that the preemption overhead must mostly depend on the number of operators required to evaluate the query Q, rather than the size of the RDF dataset.

4.2.3 Basic Graph pattern evaluation

The evaluation of a Basic Graph pattern corresponds to the evaluation of the natural join of a set of triple patterns. Physical join operators fall into three categories [26]:

- **Hash-based joins**, which store data in hash tables to join results (Symmetric Hash join [77], XJoin [71]).
- **Sort-based joins**, which take advantage of ordered relations (Merge join, ZigZag Merge join [26, 27]).
- **Loop-based joins**, which iterates over the inner relation in a loop fashion (Nested Loop join, Index Loop join [26, 27]).

Hash-based joins [27, 71, 77] operators are not suitable for preemption. As they build an in-memory hash table on one or more inputs to evaluate the join, they are full-mappings operators. However, the sort-based joins and loop joins are mapping-at-a-time operators, as they need to store only one set of solution mappings at a time in their internal state. Consequently, they are preemptible with low overhead. For sort-based joins, we can only consider merge joins, where joins inputs are already sorted on the join attribute. Otherwise, this will require to perform an in-memory sort on the inputs. In the following, we present algorithms for building preemptable Merge join and preemptable Index Loop join iterators.

Algorithm 4: A *Preemptable Merge Join Iterator MJoin*, joining the output of two iterators I_{left} and I_{right}.

Require: v: join variable, I_{left}: outer join input, I_{right}: inner join input.
Data: μ_l: last element read from I_{left}, μ_r: last element read from I_{right}.

```
1  Function Open():
2      μ_l ← nil
3      μ_r ← nil
4  Function Close():
5      I_left.Close()
6      I_right.Close()
7  Function Stop():
8      I_left.Stop()
9      I_right.Stop()
10 Function Save():
11     let s_l ← I_left.Save()
12     let s_r ← I_right.Save()
13     return ⟨s_l, s_r, μ_l, μ_r⟩
14 Function Load(s_l, s_r, μ'_l, μ'_r):
15     I_left.Load(s_l)
16     I_right.Load(s_r)
17     μ_l ← μ'_l
18     μ_r ← μ'_r
19 Function GetNext():
20     while I_left.HasNext() ∧ I_right.HasNext() do
21         if μ_l = nil then
22             μ_l ← I_left.GetNext()
23         if μ_r = nil then
24             μ_r ← I_right.GetNext()
25         if μ_l[v] = μ_r[v] then
26             non interruptible
27                 r ← μ_l ∪ μ_r
28                 μ_l ← nil
29                 μ_r ← nil
30                 return r
31         else if μ_l[v] < μ_r[v] then
32             non interruptible
33                 μ_l ← nil
34         else
35             non interruptible
36                 μ_r ← nil
37     return nil
```

Preemptable Merge join The Merge join algorithm merges the solutions mappings from two join inputs, *i.e.*, other operators that produce results sorted on the join attribute. SaGe extends the classic merge join [27] to the *Preemptable Merge join* iterator, as shown in Algorithm 4. The `GetNext()` function is very similar to its non-preemptible version [27]: the main idea is to scan both join inputs until we found a match. We use two internal variables μ_l and μ_r to save the last set of solution mappings read from the inputs I_{left} and I_{right}, respectively. When the iterator needs to read new values from the inputs, we reset the values of μ_l and/or μ_r, which triggers recursive calls to $I_{left}.GetNext()$ and $I_{right}.GetNext()$, respectively (Lines 21-24). This way, we do not wrap these calls into non-interruptible sections; otherwise, we will have to wait for their completion when preemption occurs.

The `Stop`, `Save`, and `Load` functions recursively stop, save or load the joins inputs, respectively. Thus, the only internal data structures hold by the join

operator are the two inputs and the last sets of solution mappings read from them. Propositions 7, 8, and 9 give the preemption overhead for this iterator.

Proposition 7. *The space complexity of suspending a preemptable merge join operator $MJ(P_1, P_2)$ is in $\mathcal{O}(|\operatorname{dom}(P_1)| + |\operatorname{dom}(P_2)|)$, where $\operatorname{dom}(P_i)$ is the set of variables bound in solution mappings produced by the predecessor iterator P_i.*

Proof. The `Save()` function of Algorithm 4 saves the state of both join inputs, and the last sets of solution mappings read from them. So, it space complexity is in $\mathcal{O}(|\ dom(P_1)| + |\ dom(P_2)|)$. □

Proposition 8. *The time complexity of suspending a preemptable merge join operator $MJ(P_1, P_2)$ is in $\mathcal{O}(S_{P_1} + S_{P_2})$, where S_{P_i} is the time complexity of suspending the iterator P_i.*

Proof. The non-interruptible sections of Algorithm 4 only contain assignments to local variables, so their duration is negligible. So, the time complexity of suspending the join inputs dominates the time complexity of suspending the iterator, *i.e.*, $\mathcal{O}(S_{P_1} + S_{P_2})$. □

Proposition 9. *The time complexity of resuming a preemptable merge join operator $MJ(P_1, P_2)$ is in $\mathcal{O}(R_{P_1} + R_{P_2})$, where R_{P_i} is the time complexity of resuming the iterator P_i.*

Proof. The time complexity of resuming the join inputs dominates the time complexity of the `Load()` function from Algorithm 4, *i.e.*, $\mathcal{O}(R_{P_1} + R_{P_2})$. □

Preemptable Index Loop join The Index Loop join algorithm [27] exploits indexes on the inner triple pattern for efficient join processing. This algorithm has already been used for evaluating BGPs in [36]. SaGe extends the classic Index Loop joins to a *Preemptable Index join Iterator* (PIJ-Iterator) presented in Algorithm 5. When executing, the iterator performs the same steps repeatedly until all solutions are produced: (1) It pulls solutions mappings μ_c from its predecessor. (2) It applies μ_c to tp_i to generate a *bound pattern* $b = \mu_c(tp_i)$. (3) If b has no solution mappings in $\mathcal{D}$, it tries to read again from its predecessor (jump back to Step 1). (4) Otherwise, it reads RDF triples matching b in $\mathcal{D}$, produces the associated set of solution mappings, and then goes back to Step 1.

A *PIJ-Iterator* supports preemption through the `Stop`, `Save`, and `Load` functions. The non-interruptible section of `GetNext()` only concerns a scan in the dataset. To save a PIJ Iterator, we keep the last set of solution mappings read from its predecessor and the state of the iterator used to scan the inner loop. And to resume such iterator, we resume the iterator of the inner loop where it was left

Algorithm 5: A *Preemptable Index Join Iterator* I_i: a preemptable join operator used by SaGe for BGP evaluation

Require: I_{left}: iterator responsible for the evaluation of the outer join input, tp_r: inner join input, $\mathcal{D}$: RDF dataset.
Data: μ_c: last set of solutions read from I_{left}, I_{find}: Preemptable Index Scan Iterator.

```
1  Function Open():
2      I_left.Open()
3      μ_c ← nil
4      I_find ← Preemptable Index Scan Iterator over ∅
5  Function Close():
6      I_left.Close()
7      I_find.Close()
8  Function GetNext():
9      while ¬I_find.HasNext() do
10         μ_c ← I_left.GetNext()
11         if μ_c = nil then
12             return nil
13         I_find ← Preemptable Index Scan Iterator over ⟦μ_c(tp_r)⟧_D
14     non interruptible
15         let μ ← I_find.GetNext()
16         return μ ∪ μ_c
17 Function Stop():
18     I_left.Stop()
19     I_find.Stop()
20 Function Save():
21     let s ← I_left.Save()
22     let t ← I_find.Save()
23     return ⟨s, tp_r, μ_c, t⟩
24 Function Load(s, tp', μ', t):
25     I_left.Load(s)
26     tp_r ← tp'
27     if μ' ≠ nil then
28         μ_c ← μ'
29         I_find ← Preemptable Index Scan Iterator over ⟦μ_c(tp_i)⟧_D
30         I_find.Load(t)
```

reading when preemption occurred. Propositions 11, 10, and 12 give the preemption overhead for the preemptable Index Loop join iterator.

Proposition 10. *The space complexity of suspending a preemptable Index Loop join iterator $IJoin(P, tp)$ is in $\mathcal{O}(|\operatorname{dom}(P)| + |tp| + |t|)$, where $\operatorname{dom}(P)$ is the set of variables bound in solution mappings produced by the predecessor iterator P, and $|t|$ and $|tp|$ denote the size of encoding a RDF triple and a triple pattern, respectively.*

Proof. The `Save()` function of Algorithm 5 saves the last set μ_c of solutions mappings read from the predecessor, the triple pattern tp to join with, and the saved state of the preemptable Index Scan iterator I_{left}. According to Proposition 4, the latter has a space complexity of $\mathcal{O}(|tp| + |t|)$. Thus, the space complexity of suspending a preemptable Index Loop join operator is in $\mathcal{O}(|\ dom(P)| + |tp| + |t|)$. □

Proposition 11. *The time complexity of suspending a preemptable Index Loop join iterator $IJoin(P, tp)$ is in $\mathcal{O}(S_P + \log_b |\mathcal{D}|)$, where S_P is the time complexity of suspending the iterator P.*

Proof. The non-interruptible section of the `GetNext()` function in Algorithm 5 contains a call to the `GetNext()` function of a preemptable Index scan iterator. According to Proposition 5, the duration of this section is bounded in $\mathcal{O}(\log_b |\mathcal{D}|)$. Next, the `Save()` method of Algorithm 5 saves the last RDF triple read by the inner loop, which is in constant time, and the predecessor P in the pipeline. So, if S_P is the time complexity of suspending P, then the time complexity of suspending a preemptable Index Loop join iterator is in $\mathcal{O}(S_P + \log_b |\mathcal{D}|)$. □

Proposition 12. *The time complexity of resuming a preemptable Index Loop join iterator $IJoin(P, tp)$ is in $\mathcal{O}(R_P + \log_b |\mathcal{D}|)$, where R_P is the time complexity of resuming the iterator P.*

Proof. The `Load()` method of Algorithm 5 has to resume the predecessor iterator P and resume the preemptable Index Scan iterator used to evaluate the inner loop. According to Proposition 6, the time complexity of the latter is in $\mathcal{O}(\log_b |\mathcal{D}|$. So, if R_P is the time complexity for resuming P, then the time complexity of resuming a preemptable Index Loop join iterator is in $\mathcal{O}(R_P + \log_b |\mathcal{D}|)$. □

4.2.4 UNION evaluation

A UNION operation is the union of solution mappings from two graph patterns. We consider a *multi-set union* semantic, as set unions require saving intermediate results and remove duplicates, and thus cannot be implemented as mapping-at-a-time operators. The set semantics can be restored by the Smart Web client using the DISTINCT modifier on the client-side.

Algorithm 6: A *Preemptable Multi-Set Union Iterator* $MUnion(I_1, I_2)$, merging the outputs of two iterators I_1 and I_2.

Require: v: join variable, I_1: left input, I_2: right input.
Data: μ_1: last element read from I_1, μ_2: last element read from I_2.

1 **Function** *Open()*:
2 $\quad \mu_1 \leftarrow nil$
3 $\quad \mu_2 \leftarrow nil$
4 **Function** *Close()*:
5 $\quad I_1$.Close()
6 $\quad I_2$.Close()
7 **Function** *Stop()*:
8 $\quad I_1$.Stop()
9 $\quad I_2$.Stop()
10 **Function** *Save()*:
11 $\quad$ **let** $s_1 \leftarrow I_1$.Save()
12 $\quad$ **let** $s_2 \leftarrow I_2$.Save()
13 $\quad$ **return** $\langle s_1, s_2, \mu_1, \mu_2 \rangle$
14 **Function** *Load*$(s_1, s_1, \mu'_1, \mu'_2)$:
15 $\quad I_1$.Load(s_1)
16 $\quad I_2$.Load(s_2)
17 $\quad \mu_1 \leftarrow \mu'_1$
18 $\quad \mu_2 \leftarrow \mu'_2$
19 **Function** *GetNext()*:
20 $\quad$ **if** $\mu_1 = nil \wedge I_1.HasNext()$ **then**
21 $\quad\quad \mu_1 \leftarrow I_1.GetNext()$
22 $\quad$ **else if** $\mu_2 = nil \wedge I_2.HasNext()$ **then**
23 $\quad\quad \mu_2 \leftarrow I_2.GetNext()$
24 $\quad$ **non interruptible**
25 $\quad\quad \mu \leftarrow nil$
26 $\quad\quad$ **if** $\mu_1 \neq nil$ **then**
27 $\quad\quad\quad \mu \leftarrow \mu_1$
28 $\quad\quad\quad \mu_1 \leftarrow nil$
29 $\quad\quad$ **else if** $\mu_2 \neq nil$ **then**
30 $\quad\quad\quad \mu \leftarrow \mu_2$
31 $\quad\quad\quad \mu_2 \leftarrow nil$
32 $\quad\quad$ **return** μ

Evaluating a multi-set union is equivalent to the *sequential evaluation* of all graph patterns in the union. Algorithm 6 gives the implementation of the *Preemptable Multi-set Union iterator*, which merges the output of two preemptable iterators under a multi-set semantic. It works similarly to the preemptable Merge join iterator. The `GetNext()` method sequentially produces all results from the first input I_1, and then switches to the second input I_2. As before, we use flags to notify that new sets of solution mappings must be pulled from the inputs, to avoid unnecessary non-interruptible sections. When preemption occurs, the iterator saves the state of its two inputs, and the last set of solution mappings read from them. When the iterator needs to be resumed, it reloads this information from the saved state. Propositions 14, 13, and 15 give the preemption overhead for this iterator.

Proposition 13. *The space complexity of suspending a preemptable multi-set union iterator* $MUnion(P_1, P_2)$ *is in* $\mathcal{O}(|\operatorname{dom}(P_1)| + |\operatorname{dom}(P_2)|)$, *where* $\operatorname{dom}(P_i)$ *is the set of variables bound in solution mappings produced by the predecessor iterator* P_i.

Proof. A preemptable multi-set union iterator saves the same information as a preemptable merge join iterator (Algorithm 4). Hence, following the proof of Proposition 7, the space complexity of suspending a multi-set union iterator is in $\mathcal{O}(|\ dom(P_1)| + |\ dom(P_2)|)$. □

Proposition 14. *The time complexity of suspending a preemptable multi-set union iterator* $MUnion(P_1, P_2)$ *is in* $\mathcal{O}(S_{P_1} + S_{P_2})$*, where* S_{P_i} *is the time complexity of suspending the iterator* P_i*.*

Proof. The single non-interruptible section of Algorithm 6 (Lines 24 to 32) contains assignments to local variables, so its time complexity is negligible. Next, the `Save()` function only needs to save the union inputs. So, if S_{P_i} is the time complexity of suspending the iterator P_i, then the time complexity of suspending a preemptable multi-set union iterator $MUnion(P_1, P_2)$ is in $\mathcal{O}(S_{P_1} + S_{P_2})$. □

Proposition 15. *The time complexity of resuming a preemptable union iterator* $MUnion(P_1, P_2)$ *is in* $\mathcal{O}(R_{P_1} + R_{P_2})$*, where* R_{P_i} *is the time complexity of resuming the iterator* P_i*.*

Proof. The time complexity of resuming the union inputs dominates the time complexity of the `Load()` function from Algorithm 4, *i.e.*, $\mathcal{O}(R_{P_1} + S_{P_2})$. □

4.2.5 FILTER evaluation

Algorithm 7: A *Preemptable Filter Iterator* $F(I, \mathcal{R})$

Require: $\mathcal{R}$: logical expression, $\mathcal{I}$: predecessor in the pipeline
Data: μ: set of solution mappings

1 **Function** *Open()*:
2 $I.Open()$
3 $\mu \leftarrow nil$
4 **Function** *Close()*:
5 $I.Close()$
6 **Function** *Save()*:
7 $s \leftarrow I.Save()$
8 **return** $\langle s, \mathcal{R}, \mu \rangle$
9 **Function** *Load*$(s', \mathcal{R}', \mu')$:
10 $I.Load(s')$
11 $\mathcal{R} \leftarrow \mathcal{R}'$
12 $\mu \leftarrow \mu'$
13 **Function** *GetNext()*:
14 **while** $I.HasNext()$ **do**
15 **if** $\mu \neq nil \wedge \mu \vDash \mathcal{R}$ **then**
16 **non interruptible**
17 **let** $\mu' \leftarrow \mu$
18 $\mu \leftarrow nil$
19 **return** μ'
20 **else**
21 $\mu \leftarrow I.GetNext()$

A SPARQL FILTER is denoted $F = \sigma_{\mathcal{R}}(P)$, where P is a graph pattern, and $\mathcal{R}$ is a built-in filter condition. The evaluation of F yields the solutions mappings of P that verify $\mathcal{R}$. The evaluation of some filter conditions requires to collect mappings, like the EXISTS filter, which involves the execution of a graph pattern. Consequently, we limit the filter condition to *pure logical expressions* ($=, <, \geq, \wedge$, etc) as defined in [56, 62].

Algorithm 7 shows the implementation of a *preemptable Filter operator*, which is very similar to the preemptable projection iterator. It pulls a set of solutions mappings from its predecessor in the pipeline and forwards to the next iterator in the pipeline all sets of solutions that satisfy the logical expression $\mathcal{R}$, denoted $\mu \vDash \mathcal{R}$ [56]. When preemption occurs, we save the predecessor, the last set of solutions read and the logical expression. Propositions 17, 16, and 18 give the preemption overhead for the preemptable filter iterator.

Proposition 16. *The space complexity of suspending a preemptable filter operator $F(P, \mathcal{R})$ is in $\mathcal{O}(|\operatorname{dom}(P)| + |\mathcal{R}|)$, where $\operatorname{dom}(P)$ is the set of variables bound in solution mappings produced by the iterator P and $|\mathcal{R}|$ denotes the size of encoding the logical expression $\mathcal{R}$.*

Proof. A preemptable filter operator saves the last set of solution mappings read from P and the logical expression evaluated by the iterator, so its space complexity is in $\mathcal{O}(|\,dom(P)| + |\mathcal{R}|)$. □

Proposition 17. *The time complexity of suspending a preemptable filter operator $F(P, \mathcal{R})$ is in $\mathcal{O}(S_P)$, where S_P is the time complexity of suspending the iterator P.*

Proof. The single non-interruptible section of Algorithm 7 (Lines 16 to 19) only contains assignments to local variables, so its time complexity is negligible. Next, the `Save()` function only needs to save its input, the set of solution mappings μ, and the logical expression $\mathcal{R}$. The last two can both be saved in constant time. So, if S_P is the time complexity of suspending the iterator P, then the time complexity of suspending a preemptable filter iterator $F(P, \mathcal{R})$ is in $\mathcal{O}(S_P)$. □

Proposition 18. *The time complexity of resuming a preemptable filter operator $F(P, \mathcal{R})$ is in $\mathcal{O}(R_P)$, where R_P is the time complexity of resuming the iterator P.*

Proof. The time complexity of resuming the iterator input dominates the time complexity of the `Load()` function from Algorithm 7, *i.e.*, $\mathcal{O}(R_P)$. □

4.2.6 Summary

Table 4.1 summarizes the time and space complexities of the `Suspend` and `Resume` operations for all preemptable iterators. We now compute the worst-case bounds on

these operations when applied to a pipeline of preemptable iterators, *i.e.*, a physical SPARQL query execution plan. We observe from Table 4.1 that preemptable Index Scans are the most costly preemptable iterators in regards to the preemption overhead. So, we deduce that the worst-case query for preemption is a query composed of a high number of index scans, which corresponds to a conjunctive SPARQL query evaluated using preemptable Index Loop joins.

Theorems 1, 2, and 3 give the preemption overhead for these worst-case queries. Overall, the complexity of Web preemption is higher than the initial $\mathcal{O}(|Q|)$ stated in Section 3.1. However, we demonstrate empirically in Section 4.4 that time and space complexity can be kept under a few milliseconds and kilobytes, respectively.

Theorem 1. *The space complexity of suspending a preemptable physical query execution plan is in* $\mathcal{O}(|\operatorname{dom}(Q)| + |Q| \times |tp|)$*, where* $\operatorname{dom}(Q)$ *is the set of variables bound in solution mappings produced by the evaluation of* Q*,* $|Q|$ *denotes the number of operators in the plan and* $|tp|$ *denotes the size of encoding a triple pattern.*

Proof. A worst-case query Q is a SELECT query with a single Basic Graph Pattern $B = \{tp_1, \ldots, tp_n\}$, where $n = |Q|$. To evaluate such query, we build a pipeline I of preemptable Index Loop join and Index Scans iterators such as $I = IJoin_{n-1}(tp_n, IJoin_{n-2}(tp_{n-1}, \ldots, IJoin_1(tp_2, IScan(tp_1))))$. Thus, according to Propositions 4 and 10, the space complexity of suspending I is in

$$\mathcal{O}(|\ dom(IJoin_{n-1}))| + |tp_n| + \cdots + |\ dom(IJoin_1))| + |tp_2| + |tp_1|)$$

But we know that, as we progress in the pipeline of join iterators, the number of bound variables increases, *i.e.*, $dom(P_1 \bowtie P_2) = dom(P_1) \cup dom(P_2)$. So, we have $|\ dom(IJoin_1))| \geqslant |\ dom(IJoin_2))| \geqslant \cdots \geqslant |\ dom(IJoin_{n-1})|$, which allows us to simplify the complexity as

$$\mathcal{O}(|\ dom(IJoin_{n-1})| + |tp_n| + |tp_{n-1}| + \cdots + |tp_1|) = \mathcal{O}(|\ dom(Q)| + |Q| \times |tp|)$$

□

Theorem 2. *The time complexity of suspending a preemptable physical query execution plan is in* $\mathcal{O}(|Q| \times \log_b |\mathcal{D}|)$*, where* $|Q|$ *denotes the number of operators in the plan,* b *is the order of the B+-trees used and* $\mathcal{D}$ *is the RDF dataset queried.*

Proof. As with the previous proof, we consider a pipeline I of preemptable Index Loop join and Index Scans iterators for evaluating the worst-case query. According to Propositions 5 and 11, the time complexity of suspending I is in

$$\begin{aligned}\mathcal{O}(\log_b |\mathcal{D}| + S_{IJoin_1} + \cdots + S_{IJoin_{n-1}}) &= \mathcal{O}(\log_b |\mathcal{D}| + \log_b |\mathcal{D}| + \cdots + \log_b |\mathcal{D}|) \\ &= \mathcal{O}(|Q| \times \log_b |\mathcal{D}|)\end{aligned}$$

□

Theorem 3. *The time complexity of resuming a preemptable physical query execution plan is in* $\mathcal{O}(|Q| \times \log_b |\mathcal{D}|)$*, where* $|Q|$ *denotes the number of operators in the plan,* b *is the order of the B+-trees used and* $\mathcal{D}$ *is the RDF dataset queried.*

Proof. As with the previous proofs, we consider a pipeline I of preemptable Index Loop join and Index Scans iterators for evaluating the worst-case query. According to Propositions 6 and 12, the time complexity of resuming I is in

$$\begin{aligned}\mathcal{O}(\log_b |\mathcal{D}| + R_{IJoin_1} + \cdots + R_{IJoin_{n-1}}) &= \mathcal{O}(\log_b |\mathcal{D}| + \log_b |\mathcal{D}| + \cdots + \log_b |\mathcal{D}|) \\ &= \mathcal{O}(|Q| \times \log_b |\mathcal{D}|)\end{aligned}$$

□

4.3 The SaGe Smart Web client

The SaGe approach requires a *Smart Web client* for processing SPARQL queries for two reasons. First, the client must be able to continue query execution after receiving a saved plan from the server. Second, as the preemptable server only implements a fragment of SPARQL, the smart Web client has to implement the missing operators to support full SPARQL queries. It includes SERVICE, ORDER BY, GROUP BY, DISTINCT, MINUS, FILTER EXIST, and aggregations (COUNT, AVG, SUM, MIN, MAX), but also $\bowtie, \cup, \ltimes$ and π to be able to recombine results obtained from the SaGe server. Consequently, the SaGe smart client is a *SPARQL query engine* that accesses RDF datasets through the SaGe server. Given a SPARQL query, the SaGe client parses it into a logical query execution plan, optimizes it and then builds its own physical query execution. The leafs of the plan must correspond to subqueries evaluable by a SaGe server. Figure 4.4 shows the execution plan build by the SaGe client for executing query Q_3, from Figure 4.3.

Compared to a SPARQL endpoint, processing SPARQL queries with the SaGe smart client has an overhead in terms of the number of requests sent to the server and transferred data[2]. With a SPARQL endpoint, a web client executes a query by sending a single web request and receives only query results. The smart client overhead for executing the same query is the additional number of requests and data transferred to obtain the same results. Consequently, the client overhead has two components:

- **The number of requests:** To execute a SPARQL query, a SaGe client needs $n \geq 1$ requests. We distinguish two cases: 1) The SaGe server supports the evaluation of the query, so n is equal to the number of time quantum required to process the query. Notice that the client needs to pay the network

[2]The client overhead should not be confused with the server overhead.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?name ?place WHERE {
 ?actor a dbo:Actor . # tp1
 ?actor rdfs:label ?name . # tp2
 OPTIONAL {
   ?actor dbo:birthPlace ?place . # tp3
 }
}
```

Figure 4.3: SPARQL query Q_3: finds all actors with their names and their birth places, if they exist.

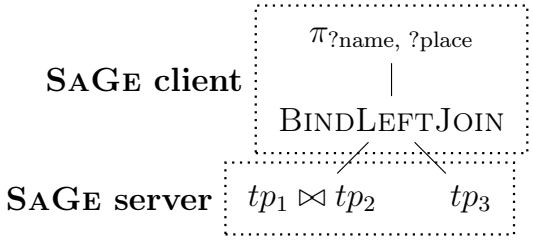


Figure 4.4: Physical query execution plan used by the SaGe smart Web client for executing query Q_3.

latency twice per request. 2) The SaGe server does not support the evaluation of the query. Then, the client decomposes the query into a set of subqueries supported by the server, evaluate each subquery as in the first case, and recombine intermediate results to produce query results.

- **Data transfer:** We also distinguish two cases: 1) If the SaGe server supports the evaluation of the query, the only overhead is the size of the saved plan S_i multiplied by the number of requests. Notice that the saved plan S_i can be returned by reference or by value, *i.e.*, saved server-side or client-side. 2) Otherwise, the client decomposes the query and recombines the results of subqueries. Among these results, some are intermediate results and part of the client overhead.

Consequently, the challenge for the smart client is to *minimize the overhead in terms of the number of requests and transferred data.* Of course, the data transfer can be reduced by using a stateful Web Preemption, but, as stated earlier, we choose a stateless implementation to preserve the scalability and fault-tolerance properties.

Algorithm 8: The OptJoin algorithm, which evaluates $[\![P_1 \rtimes P_2]\!]_{\mathcal{D}}$.

Require: P_1, P_2: SPARQL graph patterns, $\mathcal{D}$: RDF dataset, S: URL of a SaGe server.

Data: $\Omega_{P_1} \leftarrow \emptyset$, $\Omega_{P_1 \bowtie P_2} \leftarrow \emptyset$

1 **Function** *EvaluateOptJoin*(P_1, P_2, S)**:**
2 **let** $Q \leftarrow [\![(P_1 \bowtie P_2) \cup P_1]\!]_{\mathcal{D}}$
3 **let** $\Omega \leftarrow$ **Execute** Q at server S
4 **for** $\mu \in Q$ **do**
5 **if** dom $(\mu) \subseteq$ var$(P_1 \bowtie P_2)$ **then**
6 $\Omega_{P_1 \bowtie P_2} \leftarrow \Omega_{P_1 \bowtie P_2} \cup \{\mu\}$
7 **else**
8 $\Omega_{P_1} \leftarrow \Omega_{P_1} \cup \{\mu\}$
9 **return** $\Omega_{P_1 \bowtie P_2} \cup \left(\Omega_{P_1} \setminus \pi_{var(P_1)}\left(\Omega_{P_1 \bowtie P_2}\right)\right)$

We observe that the primary source of client overhead comes from the decomposition made to support full SPARQL queries, as these queries increase the number of requests sent to the server. Some decompositions are more costly than others. To illustrate, consider the evaluation of (P_1 OPTIONAL P_2) where P_1 and P_2 are expressions supported by the SaGe server. A possible approach is to evaluate $[\![P_1]\!]_{\mathcal{D}}$ and $[\![P_2]\!]_{\mathcal{D}}$ on the server, and then perform the left outer join on the client. This strategy generates only two subqueries but materializes $[\![P_2]\!]_{\mathcal{D}}$ on client. If there are no join results, $[\![P_2]\!]_{\mathcal{D}}$ are just useless intermediate results.

Another approach is to rely on a *nested loop join approach*: evaluates $[\![P_1]\!]_{\mathcal{D}}$ and for each $\mu_1 \in [\![P_1]\!]_{\mathcal{D}}$, if $\mu_2 \in [\![\mu_1(P2)]\!]_{\mathcal{D}}$ then $\{\mu_1 \cup \mu_2\}$ are solutions to $P_1 \rtimes P_2$. Otherwise, only μ_1 is a solution to the left-join. This approach sends *at least* as many subqueries to the server than there are solutions to $[\![P_1]\!]_{\mathcal{D}}$.

To reduce the communication, the SaGe client implements BindJoins to process local join in a block fashion, by sending unions of BGPs to the server. This technique is already used in federated SPARQL query processing [63] with bound joins and in BrTPF [35]. Consequently, this approach reduces the number of requests sent to the SaGe server by a factor equivalent to the size of a block of mappings. However, the number of requests sent still depends on the cardinality of P_1.

Consequently, we propose a new technique, called OptJoin, for optimizing the evaluation of a subclass of left-joins, *i.e.*, SPARQL queries with OPTIONAL clauses. The approach relies on the fact that:

$$[\![P_1 \rtimes P_2]\!]_{\mathcal{D}} = [\![P_1 \bowtie P_2]\!]_{\mathcal{D}} \cup \left([\![P_1]\!]_{\mathcal{D}} \setminus [\![\pi_{var(P_1)}\left(P_1 \bowtie P_2\right)]\!]_{\mathcal{D}}\right)$$

So we can deduce that: $[\![P_1 ⟕ P_2]\!]_{\mathcal{D}} \subseteq [\![(P_1 \bowtie P_2) \cup P_1]\!]_{\mathcal{D}}$.

Algorithm 8 presents the OptJoin algorithm. If both P_1 and P_2 are evaluable by the SaGe server, then the left-join is computed as follows. First, it sends the query $(P_1 \bowtie P_2) \cup P_1$ to the server (Line 3). Then, for each mapping μ received, it builds local materialized views for $[\![P_1 \bowtie P_2]\!]_{\mathcal{D}}$ and $[\![P_1]\!]_{\mathcal{D}}$ (Lines 4 to 8). The client knows that $\mu \in [\![P_1 \bowtie P_2]\!]_{\mathcal{D}}$ if $dom\ (\mu) \subseteq var\ (P_1 \bowtie P_2)$ (Line 5), otherwise $\mu \in [\![P_1]\!]_{\mathcal{D}}$. Finally, the client uses the views to process the left-join locally (Line 9). With this technique, the client only uses one subquery to evaluate the left-join and, in the worst case, it transfers $[\![P_1 \bowtie P_2]\!]_{\mathcal{D}}$ as additional intermediate results.

To illustrate, consider query Q_3 from Figure 4.3. Q_3, with 88334 solutions. The cardinality of $tp_1 \bowtie tp_2$ is also of 88334, as every actor has a birthplace. It is the worse case for a BindLeftJoin, which will require $\frac{88334}{Block\ size}$ additional requests to evaluate the left join. However, with an OptJoin, the client executes Q_3 using approximately 500 requests.

We implement both BindLeftJoin and OptJoin as physical operators to evaluate OPTIONALs, depending on the query. We also implement regular BindJoin for processing SERVICE queries.

4.4 Experimental study

We want to empirically answer the following questions: What is the overhead of Web preemption in time and space? Does Web preemption improve the average workload completion time? Does Web preemption enhance the time for the first results? What are the client overheads in terms of numbers of requests and data transfer? We use Virtuoso as the baseline for comparing with SPARQL endpoints, with TPF and BrTPF as the baselines for the LDF approach.

We implemented the SaGe client in Java, using Apache Jena[3]. As an extension of Jena, SaGe is just as compliant with SPARQL 1.1. The SaGe server is implemented as a Python Web service and uses HDT [24] (v1.3.2) for storing data. Note that the current implementation of HDT cannot ensure $log_b(n)$ access time for all triple patterns, like $(?s\ p\ ?o)$. This impacts the performance of SaGe negatively when resuming some queries. The code and the experimental setup are available on the companion website[4].

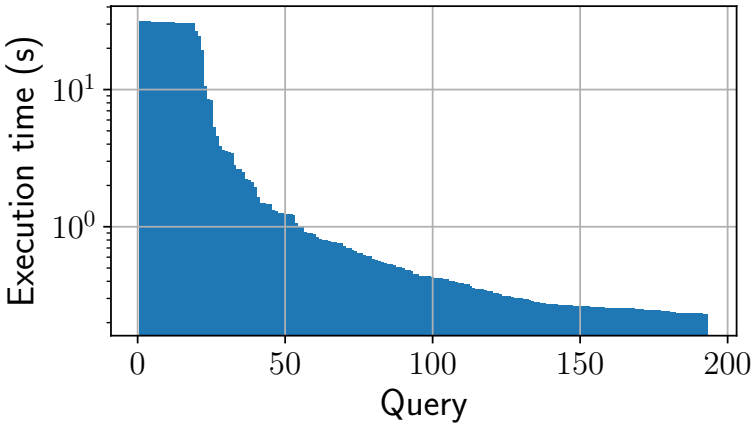


Figure 4.5: Distribution of query execution time.

4.4.1 Experimental setup

Dataset and Queries: We use the Waterloo SPARQL Diversity Benchmark (WatDiv[5]) [4]. We re-use the RDF dataset and the SPARQL queries from the BrTPF [35] experimental study[6]. The dataset contains 10^7 triples, and we arrange queries in 50 workloads of 193 queries each. They are SPARQL conjunctive queries with STAR, PATH, and SNOWFLAKE shapes. They vary in complexity, up to 10 joins per query with very high and very low selectivity. 20% of queries are executed in more than $\approx 30s$ to be executed using the Virtuoso server. All workloads follow nearly the same distribution of query execution times, as presented in Figure 4.5, where we measured the query execution times for one workload of 193 queries with SaGe and an infinite time quantum.

Approaches: We compare the following approaches:

- SaGe: We run the SaGe query engine with various time quantums: 75ms and 1s, denoted SaGe-75ms and SaGe-1s, respectively. HDT indexes are loaded in memory while HDT data is stored on disk.
- *Virtuoso:* We run the Virtuoso SPARQL endpoint [23] (v7.2.4), *without any quotas or limitations.*
- *TPF:* We run the standard TPF client (v2.0.5) and TPF server (v2.2.3) with HDT files as backend (same settings as SaGe).

[3] https://jena.apache.org/
[4] https://github.com/sage-org/sage-experiments
[5] http://dsg.uwaterloo.ca/watdiv/
[6] http://olafhartig.de/brTPF-ODBASE2016

- *BrTPF:* We run the BrTPF client and server used in [35], with HDT files as backend (same settings as SaGe). BrTPF is currently the LDF approach with the lowest data transfer [35].

Servers configurations: We run all the servers on a machine with Intel(R) Xeon(R) CPU E7-8870@2.10GHz and 1.5TB RAM.

Clients configurations: To generate load over servers, we rely on 50 clients, each one executing a different workload of queries. All clients start executing their workload simultaneously. The clients access servers through HTTP proxies to ensure that client-server latency is around 50ms.

Evaluation Metrics: Presented results correspond to the average obtained of three successive execution of the queries workloads.

- *Workload completion time (WCT)*: is the total time taken by a client to evaluate a set of SPARQL queries, measured as the time between the first query starting and the last query completing.
- *Time for first results (TFR)* for a sage:query: is the time between the query starting and the production of the first query's results.
- *Time preemption overhead*: is the total time taken by the server's `Suspend` and `Resume` operations.
- *The number of HTTP requests*: is the total number of HTTP requests sent by a client to a server when executing a SPARQL query.
- *The data transfer*: is the quantity of transferred bytes when executing a SPARQL query.

4.4.2 Experimental results

We first ensure that the SaGe approach yields complete results. We run both Virtuoso and SaGe and verify that, for each query, SaGe delivers complete results using Virtuoso results as ground truth.

What is the overhead in time of Web preemption? The overhead in time of Web preemption is the time spent by the Web server for suspending a running query and the time spent for resuming the next waiting query. To measure the overhead, we run one workload of queries using SaGe-75ms and measure time elapsed for the `Suspend` and `Resume` operations. Figure 4.6 shows the overhead in

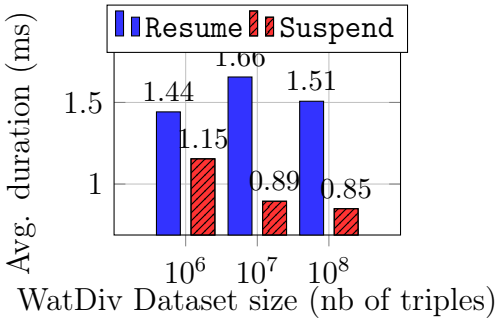


Figure 4.6: Average preemption overhead.

Mean	Min	Max	Standard deviation
1.716 kb	0.276 kb	6.212 kb	1.337 kb

Table 4.2: Space of saved physical query execution plans.

time for SaGe `Suspend` and `Resume` operations using different sizes of the WatDiv dataset. Generally, the size of the dataset does not impact the overhead, which is around $1ms$ for `Suspend` and $1.5ms$ for `Resume`. As expected, the overhead is greater for the `Resume` operation than the `Suspend` operation, due to the cost of resuming Index scans in the plan. With a quantum of 75ms, the overhead is $\approx 3\%$ of the quantum, which is negligible.

What is the overhead in space of Web preemption? The overhead in space of the Web preemption is the size of saved plans produced by the `Suspend` operation. According to Section 4.2, we determined that the SaGe physical query plans can be saved in $O(|Q|)$. To measure the overhead, we run a workload of queries using SaGe-75ms and measure the size of saved plans. We compress saved plans using Google Protocol Buffers[7]. Table 4.2 shows the overhead in space for SaGe. As we can see the size of a saved plan remains very small, with no more than 6 kb for a query with ten joins. Hence, this space is indeed proportional to the size of the plan suspended.

Does Web preemption improve the average workload completion time? The SaGe server has a restricted choice of physical query operators, so physical query execution plans generated by the SaGe server should be less performant than those produced by Virtuoso. This tradeoff only makes sense if the Web preemption compensates for the loss in performance. Compensation is possible only if the workload alternates long-running and short running queries. In the setup, each

[7] https://developers.google.com/protocol-buffers/

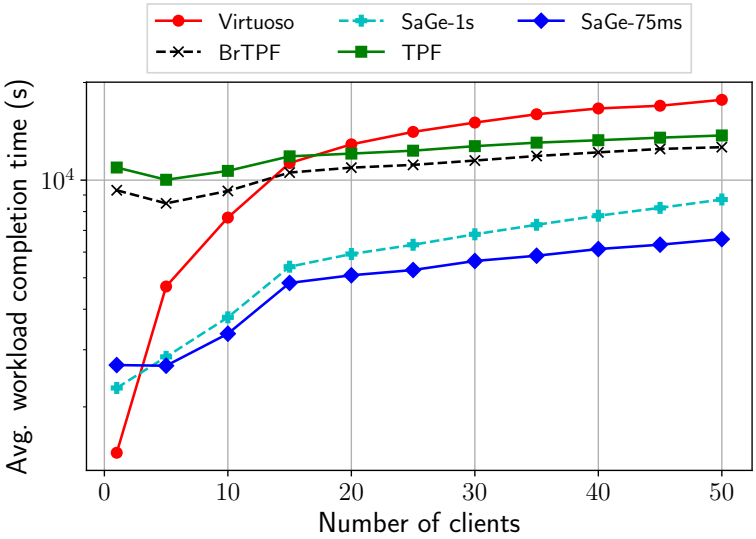


Figure 4.7: Average workload completion time per client, with up to 50 concurrent clients (logarithmic scale).

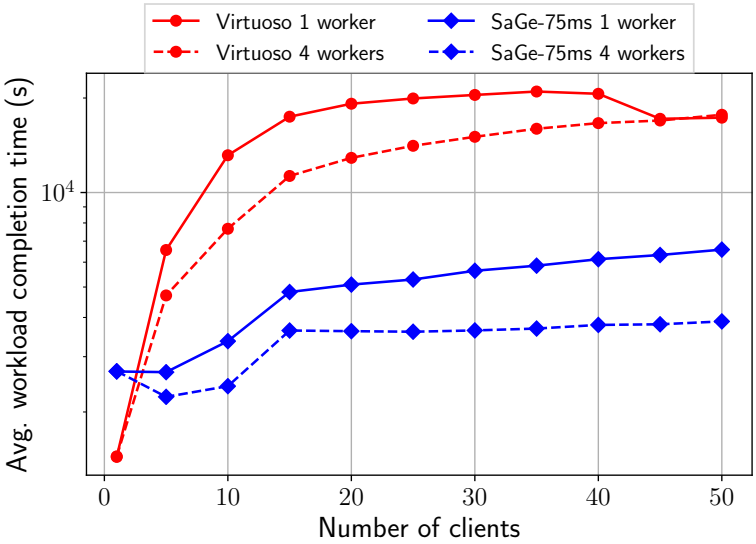


Figure 4.8: Average workload completion time per client, with 4 workers (logarithmic scale).

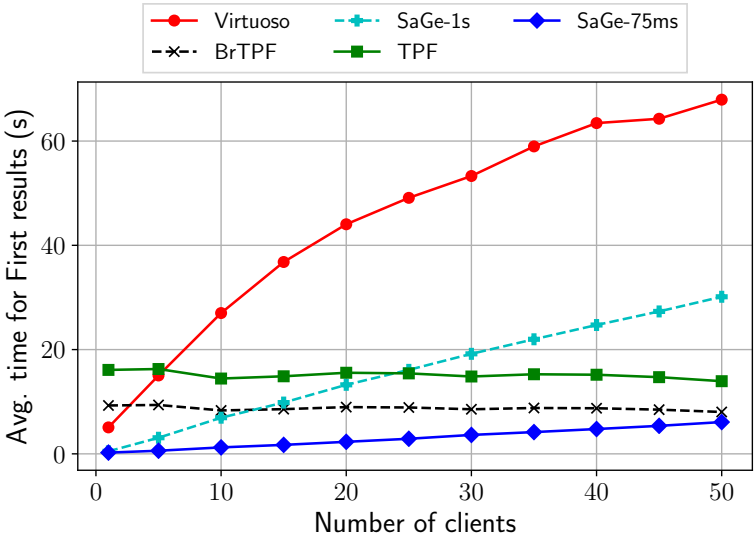


Figure 4.9: Average time for first results (over all queries), with up to 50 concurrent clients (linear scale).

client runs a different workload of 193 queries that vary from 30s to 0.22s, following an exponential distribution. All clients execute their workload concurrently and start simultaneously. We experiment up to 50 concurrent clients by step of 5 clients. As there is only one worker on the Web server and queries execution times vary, this setup is the worst case for Virtuoso.

Figure 4.7 shows the average workload completion time obtained for all approaches, *with a logarithmic scale*. As expected, the convoy effect significantly impacts Virtuoso, and it delivers the worse WTC after 20 concurrent clients. TPF and BrTPF avoid the convoy and behave similarly. BrTPF is more performant thanks to its bind-join technique that group requests to the server. SaGe-75ms and SaGe-1s avoid the convoy effect and delivers better WTC than TPF and BrTPF. As expected, increasing the time quantum also increases the probability of convoy effect, and, globally, SaGe-75ms offers the best WTC. We rerun the same experiment with four workers for SaGe-75ms and Virtuoso. Figure 4.8 shows the average workload completion time obtained for both approaches. As we can see, both approaches benefit from the four workers. However, Virtuoso still suffers from the convoy effect.

Does Web preemption improve the time for the first results? The Time for first results (TFR) for a query is the time between the query starting and the production of the first query's results. Web preemption should provide better a time for the first results. Avoiding the convoy effect allows us to start queries earlier

Dataset	Virtuoso	SaGe-1s	SaGe-75ms	BrTPF	TPF
WatDiv 10^7	193	645	4 082	$9,2 \cdot 10^4$	$2,55 \cdot 10^5$
FEASIBLE	166	1 822	3 305	$2,95 \cdot 10^4$	$1,86 \cdot 10^5$

Table 4.3: Average number of HTTP requests sent to server with WatDiv and FEASIBLE-DBpedia datasets.

and then get results earlier. We rerun the same setup as in the previous section and measure the time for the first results (TFR). Figure 4.9 shows the results *with a linear scale*. As expected, Virtuoso suffers from the convoy effect that degrades the TFR significantly when the concurrency increases. TPF and BrTPF do not suffer from the convoy effect, and TFR is stable over concurrency. The main reason is that delivering a page of result for a single triple pattern takes $\approx$ 5ms in our experiment, so the waiting time on the TPF server grows very slowly. BrTPF is better than TPF due to its bind-join technique. The TFR for SaGe-75ms and SaGe-1s increases with the number of clients and the slope seems proportional to the quantum, because the waiting time on the server increases with the number of clients, as seen previously. Reducing the quantum improves the TFR, but increases the number of requests and thus deteriorates the WTC.

What are the client overheads in terms of the number of requests and data transfer? The client overhead in requests is the number of requests that the smart client sent to the server to get complete results minus one, as Virtuoso executes all queries in one request. As WatDiv queries are pure conjunctive queries and supported by the SaGe server, the number of requests to the server is the number of time quantum required to evaluate the whole workload. We measure the number of requests sent to servers with one workload for all approaches, with results shown in Table 4.3. As expected, Virtuoso just sends 193 requests to the server. SaGe-75ms sends 4082 requests to the server, while TPF sends 2.55×10^5 requests to the TPF server. We can see also that increasing the time quantum significantly reduces the number of requests; SaGe-1s sends only 645 requests. However, this seriously deteriorates the average WCT and TFR, as presented before. To compute the overhead in data transfer of SaGe, we just need to multiply the number of requests by the average size of saved plans; for SaGe-75ms, the client overhead in data transfer is $4082 \times 1,3$ kb $= 5.45$ Mo. As the total size of the results is 51Mo, the client overhead in data transfer is $\approx 10\%$. For TPF, the average size of a page is 7ko; $2.5 \cdot 10^5 \times 7k = 1.78$ Go, so the data transfer overhead is $\approx 340\%$ for TPF.

What are the client overheads in terms of numbers of requests and data transfer for more complex queries? The WatDiv benchmark does not

Time quantum	SaGe+BindLeftJoin	SaGe+OptJoin
75ms	72 489	5 656
1s	70 964	511

Table 4.4: Comparison of the average number of HTTP requests sent to server when using the BindLeftJoin and OptJoin operators

generate queries with OPTIONAL or FILTER operators. If the SaGe server supports some filters, the OPTIONAL operator and other filters impact the number of requests sent to the SaGe server, as explained in Section 4.3. First, we run an experiment to measure the number of requests for queries with OPTIONAL. We generate new WatDiv queries from one set of 193 queries, using the following protocol. For each query $Q = \{tp_1, \ldots, tp_n\}$, we select $tp_k \in Q$ with the highest cardinality, then we generate $Q' = tp_k \bowtie (Q \setminus tp_k)$. Such queries verify conditions for using BindLeftJoin and OptJoin. They are challenging for the BindLeftJoin as they generate many mappings; they are also challenging for the OptJoin as all joins yield results. Table 4.4 shows the results when evaluating OPTIONAL queries with OptJoin and BindLeftJoin approaches. We observe that, in general, OptJoin outperforms BindLeftJoin. Furthermore, OptJoin is improved when using a higher quantum, as the single subquery sent is evaluated more quickly. It is not the case for BindLeftJoin, as the number of requests still depends on the number of intermediate results.

Finally, we re-use 166 SELECT queries from Feasible [59] with the DBpedia 3.5.1 dataset, generated from real-users queries. We excluded queries that time-out, identified in [59], and measure the number of requests sent to the server. Table 4.3 shows the results. Of course, Virtuoso just send 166 requests to the server. We observe that the ratio of requests between SaGe-75 and TPF is nearly the same as the previous experiment. However, the marge between SaGe-1s and SaGe-75ms decrease, because most requests sent are produced by the decomposition of OPTIONAL and FILTER and not by the evaluation of BGPs.

Chapter 5

Conclusion

5.1 Summary of contributions

In this thesis, we choose to tackle the issue of building public SPARQL query servers that allows any data consumers to execute any SPARQL query with complete results while remaining available. Existing approaches ensure availability using a quota policy that restricts query execution. By following an approach based on time-sharing, similar to those used in operating systems, we made the following hypothesis

> We hypothesize that the issue related to quotas is not interrupting a query, but the impossibility for the client to *resume* the query execution afterward.

Based on this, we proposed a new query execution model called `Web Preemption`. Web preemption is the capacity of a Web server to suspend a running query after a time quantum with the intention to resume it later. When suspended, the server sends the saved state S_i of the query to the Web client. Then, the client can resume query execution by sending S_i back to the Web server. Compared to existing query execution models, Web Preemption ensures *a fair allocation of Web server resources across queries, a better average query completion time per query, and a better time for first results.*

This approach is original, as no other approach tackles the issue of time-sharing at the scale of SPARQL queries running in a public server. However, our new model adds an overhead for the Web server to suspend the running query and resume the next waiting query. Consequently, the next scientific challenge is to keep this overhead marginal, whatever the running SPARQL queries, to ensure good query execution performance.

SaGe is a SPARQL query engine that implements the Web Preemption model. To minimize the preemption overhead, SaGe distributes SPARQL query execution between a preemptive SPARQL query server and a smart Web client. The first supports the evaluation of all operators that can be suspended and resumed with a low preemption overhead, while the second supports the evaluation of all remaining operators. This way, SaGe supports allows for the execution of any SPARQL 1.1 query. Compared to SPARQL endpoint approaches without quotas, SaGe avoids the convoy effect and is a winning bet as soon as the queries of the workload vary in execution time. Compared to LDF approaches [35, 72], SaGe offers a more expressive server interface with join, union, and filter evaluated on the server side. Consequently, it considerably reduces data transfer and the communication cost, improving the execution time of SPARQL queries.

5.2 Perspective works

This section details the perspective works following the contributions of this thesis.

5.2.1 Updating Knowledge Graphs under the Web preemption model

A public SPARQL query server can receive two types of queries: **read-only** queries sent by data consumers to query the RDF dataset, and **read-write** queries sent by the data provider to update the hosted RDF data. In SaGe, we made the hypothesis that the server only receives read-only queries, but we could extend our model to supports the concurrent evaluation of both types of SPARQL queries. In database systems, the concurrent execution of read-only and read-write queries can lead to *data consistency issues*: if two queries are updating the same RDF triples, what happens? It is the goal of the database's *concurrency control system* [14] to handle these issues, by imposing a set of constraints, called *consistency criteria*, that dictate how concurrent queries are executed [57]. A query whose execution violates these constraints generates a *conflict* and is often *aborted* and rescheduled for execution. The most common consistency criterion is *serializability* [14, 55], implemented using database *transactions*. Each query is wrapped inside a transaction, and while it is executing, its results are not available to other concurrent queries. When a transaction completes, the concurrency control system invokes an algorithm called *atomic commitment* to ensure that all operations made by the transaction can be applied as a *single unit of computation* to the data.

In the context of Web Preemption, concurrency control is a challenging issue because **it is impossible to suspend the atomic commitment algorithm**. Indeed, this algorithm must run as a *non-interruptible section*, as it acquires *locks* on

each data item that will be updated by the transaction. While the lock granularity can be very small [44, 54], a transaction always needs to acquire locks, and it will only release them when it completes. So, locks cannot be released when preemption occurs, which generates convoy effects for long-running transactions, as reported by Blasgen et al. in [16].

We think that this issue can be solved by performing two modifications to the Web preemption model. First, we think that using two distinct pool of workers, one for each type of query, can remove convoy effects between read-write and read-only queries. Next, we need to ensure that read-only queries can always read *a consistent snapshot of the database*. However, a concurrent query might delete the information required to resume preemptable iterators. We think that a new *multi-version concurrency control* algorithm [14, 54] can be developed, to ensure consistent reads under the Web preemption model.

5.2.2 Web Preemption for Web-querying languages

In this work, we implemented the Web Preemption model for the evaluation of SPARQL queries. But we could consider the execution of queries in other languages under the same model. GraphQL [39] is a recently proposed, and increasingly adopted, framework for providing a new type of data access interface on the Web. A core component of the GraphQL framework is a query language for expressing the data retrieval requests issued to GraphQL-aware Web servers, as in a graph-oriented database. However, as reported in [39], GraphQL query servers also face the issue of convoy effects and solve it using quotas. For example, Github enforces severe limitations on its GraphQL API [1] and does not allow for complex queries or more than 5 000 authenticated requests per hour. We think that a preemptive GraphQL server could solve these issues like SaGe did for SPARQL queries. The challenges are the same as in SaGe: build *preemptive GraphQL physical query operators* to minimize the preemption overhead. Since most operators of the GraphQL language have SPARQL equivalents [68], we think that we can obtain results close to those of SaGe.

5.2.3 Optimize client-side query processing for Smart Web clients

The SaGe Smart Web Client acts as a client-side SPARQL query engine that distributes query processing between the server and the client. In this thesis, we do not look at some unusual challenges regarding this execution workflow. First, we could extend the SPARQL optimizer used by the client to compute the optimal

[1] https://developer.github.com/v4

distribution of query processing between the client and the server, *i.e.*, which sub-plans to ship to the SaGe server to minimize data transfers and SPARQL query execution times. It is a challenging issue because the client does not have access to the same statistics as the SaGe server. So, it will have to download histograms from the server, which increases the data transferred. To solve it, we could rely on *adaptive query processing techniques* [10, 20] for client-side SPARQL processing. Federated SPARQL query processing [3] and smart LDF clients [2] successfully applied similar methods to perform on-the-fly query optimization without having to download too much statistics from the server.

Next, we could propose new rewriting techniques, like the OptJoin technique, to optimize the evaluation of full-mappings SPARQL operators. Aggregations operators (`GROUP BY`, `COUNT`, `AVG`, ...) are good candidates, as they are crucial to compute statistics over the Linked Data using SPARQL queries [41, 66]. We think that an approach similar to Map-Reduce [19, 45] is feasible to compute SPARQL aggregations more efficiently. The SaGe server will only compute *partial aggregations* over solutions mappings produced during a quantum. Then the smart client will *merge partial results* using client-side reducing functions to produce the final query results. This approach could drastically reduce the data transferred and the number of HTTP requests, speeding up the overall SPARQL query execution. However, *distinct aggregations* might create issues in terms of data transfer, as they need to deduplicate data beforehand, which requires to materialize the complete query results.

Appendix A

Résumé en langue française

A.1 Introduction

A.1.1 Motivations

Le Web des Documents [12] est basé sur trois principes fondateurs. Le premier est l'URI, qui associe un identifiant unique à chaque ressource du Web. La forme la plus commune d'URI est l'URL, *e.g.*, `https://www.univ-nantes.fr` est l'URL de la page d'accueil du site web de l'Université de Nantes. Le deuxième principe est le protocole HTTP, qui permet à tout utilisateur d'accéder à une URL et d'y télécharger le document associé, ou alors d'obtenir un code d'erreur. Le troisième et dernier principe est le langage HTML, un format hypertexte utilisé pour décrire le contenu des pages Web.

Cependant, les documents HTML classiques ne sont pas suffisamment expressifs pour permetre de décrire des entités et leurs propriétés sans aucune ambiguité [15]. Pour ce faire, le Web sémantique [13] étend le Web des documents pour créer un *Web des données*, où les documents texte sont remplacés par des données structurées qui peuvent être automatiquement analysés par des ordinateurs. Ce nouveau Web ré-utilise les deux premiers principes fondateurs du Web des documents (les URLs et le protocole HTTP), mais remplace l'usage de documents HTML par des documents RDF [18]. La structure du modèle de données RDF [18] est basée sur des *triplets RDF*. Un triplet est composé d'un *sujet*, d'un *prédicat* et d'un *objet* : il énonce un fait à propos d'une entité du Web. En suivant les principes du Linked Open Data (LOD), proposés en 2006 par Tim Berners-Lee [11], les fournisseurs de données ont publié des milliards de documents RDF sur le Web [15, 61]. La plupart sont hébergés sous la forme de simples fichiers RDF, qui peuvent être téléchargés via leurs URIs. L'URI `http://dbpedia.org/resource/Neil_Gaiman` permet d'accéder à l'ensemble des triplets RDF sur l'auteur Neil Gaiman.

Cependant, l'hébergement de fichiers RDF réduit grandement la capacité des

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX dbr:  <http://dbpedia.org/resource/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?author ?book WHERE {
  ?author rdf:type dbo:Person .
  ?author foaf:name ?name .
  ?book dbo:author ?author .
}
```

Figure A.1: Une requête SPARQL qui cherche tous les auteurs et leurs livres dans le jeu de données DBpedia.

utilisateurs à répondre à des questions complexes en utilisant le Web des données. Par exemple, si nous voulons obtenir la liste de tous les acteurs de cinéma, nous devons télécharger l'ensemble des documents RDF qui contiennent des triplets intéressants, ce qui est une opération complexe et très coûteuse en temps. Pour pallier à ce problème, une collection de documents RDF peut également être hébergée par *un SPARQL endpoint*, un système de bases de données spécialisé qui permet d'exécuter des requêtes complexes sur des triplets RDF. Les utilisateurs peuvent envoyer, via le protocole HTTP, des requêtes SPARQL, qui sont des requêtes type SQL qui permettent à l'utilisateur d'interroger des documents RDF via un langage de requête structuré. Par exemple, le DBpedia SPARQL endpoint [1] permet à n'importe qui d'exécuter des requêtes sur le jeu de données DBpedia, qui contient des milliards de faits récupérés depuis Wikipedia [9]. La figure A.1 montre une requête SPARQL qui trouve tous les auteurs et leurs livres dans DBpedia.

Les services publics d'évaluations de requêtes SPARQL, comme les SPARQL endpoints, sont très importants pout le Web des données, car ils permettent de développer des applications qui ré-utilisent les milliards de triplets RDF disponibles dans le LOD. Par exemple, les SPARQL endpoints de DBpedia [9] et Wikidata [73] sont tous les deux utilisés comme bases de connaissances par des systèmes questions-réponses multilingues [22, 69] et des agents conversationnels [8]. Cependant, *fournir un service public qui permet à n'importe quel utilisateur d'exécuter n'importe quelle requête SPARQL est encore un problème ouvert* [6]. Étant donné que ces services sont soumis à une charge imprévisible de requêtes SPARQL, le défi est d'assurer qu'ils restent *disponibles* malgré des variations dans les taux d'arrivée des requêtes et des *ressources* à leur disposition. Les fournisseurs de données se réfèrent souvent à ce défi comme *la mise en place d'une politique d'accès équitable*.

[1] http://dbpedia.org/sparql

Pour mettre en place ce type de politique d'accès, la plupart des SPARQL endpoints sont configurés avec *des politiques de quotas* qui garantissent un partage équitable des ressources entre les utilisateurs. Si une requête SPARQL dépasse l'un de ces quotas, elle est interrompue par le serveur et l'utilisateur qui l'a émise reçoit une erreur. Il existe de nombreux types de quotas, mais les restrictions les plus communes concernent: 1) Le temps d'exécution des requêtes SPARQL, 2) Le nombre de résultats obtenus par requête, et 3) Le taux d'envoi de requêtes par adresse IP. Par exemple, le SPARQL endpoint de DBpedia ne permet pas d'exécuter des requêtes pendant plus de 120 secondes ou qui obtiennent plus de 10000 résultats [2]. De plus, il ne permet pas plus de 50 connexions parallèles et plus de 100 requêtes HTTP par seconde par adresse IP.

Certains de ces quotas ne sont pas perturbants pour les utilisateurs, *e.g.*, si le taux d'arrivée des requête est restreint, alors il suffit de réessayer plus tard. Cependant, les limites sur les temps d'exécution ou le nombre de résultats obtenus conduisent les services d'évaluation de requêtes à ne délivrer que des *résultats partiels. Il s'agit d'une limite importante à l'utilisation de ces services*, car il devient alors impossible de développer des applications basées dessus [58].

Cependant, sans ces politiques de quotas, les SPARQL endpoints deviennent sujet à un phénomène appelé *effet convoi* [16] car ils exécutent les requêtes entrantes en suivant une politique First-Come First-Served (FCFS) [25]. Commes les requêtes sont exécutées dans leur ordre d'arrivée au serveur, une requête longue peut occuper l'intégralité des ressources du serveur et empêcher les autres requêtes, même courtes, de s'exécuter.

Dans l'état actuel du LOD, nous somme donc face à un dilemne. D'un côté, nous pouvons construire un service d'évaluation de requêtes SPARQL performant grâce aux politiques de quotas, mais il peut fournir des résultats incomplets aux utilisateurs. De l'autre, nous pouvons construire un service qui fournit toujours des résultats complets, mais dont la disponibilité n'est pas garantie du fait des effets convois. Aucune de ces deux solutions n'est acceptable pour développer des applications basées sur le Web des données, car les développeurs ont besoin de services à la fois disponibles et qui délivrent toujours des résultats complets.

A.1.2 Contributions et contenu de cette thèse

Dans cette thèse, nous proposons de résoudre le problème relatif à *la construction des services publics d'évaluation de requêtes SPARQL qui permettent à n'importe quel utilisateur d'exécuter n'importe quelle requête SPARQL en obtenant des résultats complets*. Pour ce faire, nous proposons deux contributions. La première est un nouveau modèle d'exécution pour les requêtes SPARQL, dénommé *la préemption*

[2] http://wiki.dbpedia.org/public-sparql-endpoint

Web, qui s'inspire des travaux sur les processeurs à temps partagé [5]. La préemption Web est la capacité d'un serveur Web à suspendre l'exécution d'une requête SPARQL après un temps donné, avec l'intention d'en reprendre l'exécution ultérieurement. Une fois suspendu, l'état S_i de l'exécution est renvoyé au client qui a émis la requête, et il pourra en reprendre l'exécution en renvoyant S_i au serveur. Comparée aux modèles existants, la préemption Web permet de garantir *une allocation équitable des ressources du serveur entre les utilisateurs, un meilleur temps moyen d'exécution des requêtes et un meilleur temps d'obtention des premiers résultats.*

Cependant, notre nouveau modèle d'exécution ajoute un surcoût au Web serveur, pour suspendre une requête en cours d'exécution et reprendre l'exécution de la suivante. Donc, le défi scientifique est de maintenir ce surcoût au minimum, pour garantir les meilleurs performances d'exécution des requêtes SPARQL. A ces fins, nous proposons **SaGe**, un moteur préemptif d'évaluation de requêtes SPARQL qui implémente le modèle de la préemption Web. Nous définissons un ensemble *d'opérateurs interruptibles*, utilisés pour évaluer les requêtes SPARQL, pour lesquels nous bornons le surcoût de préemption. Ces opérateurs permettent de construire un serveur Web préemptif qui supporte l'exécution d'une grande partie du langage de requête SPARQL. Étant donné que tous les opérateurs n'ont pas un surcoût de préemption raisonnable, nous divisons le moteur SaGe en deux composants : un *serveur SPARQL préemptif*, qui utilise nos opérateurs interruptibles pour exécuter des requêtes SPARQL selon le modèle de la préemption Web, et un *client intelligent*, qui exécute côté client les opérations non supportées par le serveur. En combinant ces deux composants, nous permettons l'évaluation de l'ensemble du langage de requête SPARQL selon notre nouveau modèle d'exécution.

Enfin, nous proposons une étude expérimentale complète du moteur SaGe (client et serveur), pour le comparer aux approches existantes utilisées pour construire des services publics d'évaluation de requêtes SPARQL. Nos résultats expérimentaux démontrent que SaGe est plus performant que les approches existantes, aussi bien en termes de temps moyen d'exécution des requêtes que de temps d'obtention des premiers résultats.

A.2 Le modèle d'exécution de la Préemption Web

Le modèle d'exécution de la préemtpion Web se définit formellement comme suit. Nous considérons un *serveur Web préemptif*, qui héberge un jeu de données RDF *en lecture seule*, et un *client Web intelligent*, qui évalue des requêtes SPARQL en utilisant le serveur. Ce dernier possède un ensemble de *workers*, tous paramétrés avec un même quantum de temps, qui sont chargés d'exécuter les requêtes. Le serveur possède également une *file d'attente* qui enregistre les requêtes entrantes lors tous les workers sont occupés. Nous considérons une population *infinie* de

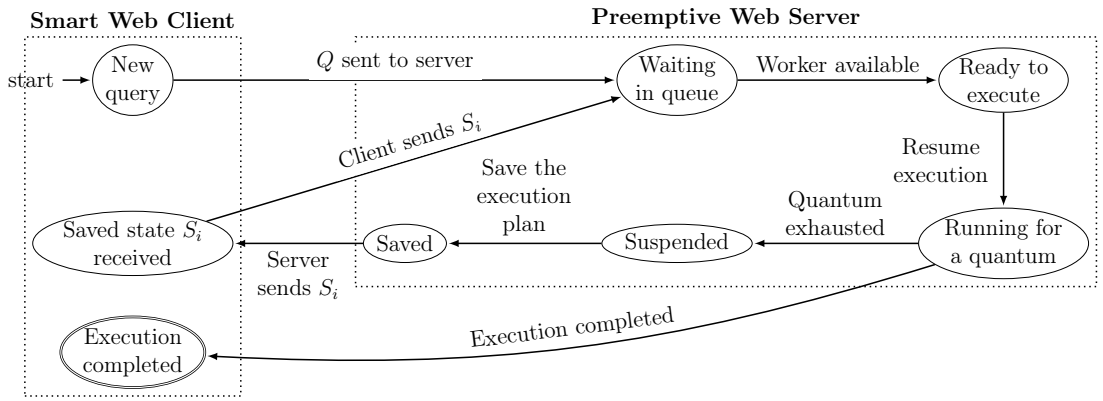


Figure A.2: États possibles de l'exécution d'une requête avec la préemption Web.

clients, une file d'attente de taille *finie* et un nombre *fini* de workers.

Un serveur Web préemptif supporte trois opérations: `Execute`, `Resume`, and `Suspend`. L'opération `Execute` est utilisée pour exécuter une requête Q pour un quantum de temps q. Elle prend en entrée une requête, représentée par *son plan d'exécution* denoté P_{Q_i}, et l'exécute pour une durée égale à q. Ensuite, elle produit *un ensemble de solutions* ω_i et le plan d'exécution P'_{Q_i}, qui est l'état de P_{Q_i} après la fin de l'exécution ($\texttt{Execute}(P_{Q_i}) = \langle \omega_i, P'_{Q_i} \rangle$). L'opération `Suspend`, qui sauvegarde l'état d'exécution d'une requête, est appliqué sur P_{Q_i} et produit *un état sauvegardé* S_i ($\texttt{Suspend}(P_{Q_i}) = S_i$). L'opération `Resume`, qui reprend l'exécution d'une requête depuis un état sauvegardé, est l'action inverse : elle s'applique sur S_i et restaure le plan d'exécution ($\texttt{Resume}(S_i) = P_{Q_i}$).

La figure A.2 présente les différents états d'exécution possible d'une requête dans ce modèle. Les transitions sont effectuées par le serveur Web ou par le client intelligent. Le serveur accepte dans sa file d'attente des requêtes Web qui contiennent des requêtes SPARQL ou des états sauvegardés. Si un worker est disponible, alors il récupère une requête depuis la file d'attente. S'il s'agit d'une requête SPARQL Q_i, il produit un plan d'exécution P_{Q_i}, selon le paradigme "optimize-then-execute" [27], puis l'exécute pour un quantum de temps. S'il s'agit d'un état sauvegardé S_i, le worker reprend l'exécution de la requête correspondante en utilisant l'opération `Resume`. Le temps nécessaire pour reprendre une requête n'est pas déduit du quantum.

Lorsque l'exécution d'une requête est terminée avant la fin du quantum, le serveur renvoie les résultats produits au client, pusi l'exécution est terminée. En revanche, si le quantum est épuisé mais que l'exécution n'est pas achevée, alors le serveur utilise l'opération `Suspend` pour interrompre l'exécution et produire un état sauvegardé S_i. Le temps nécessaire pour cette interruption n'est pas déduit du quantum. Ensuite, le serveur envoie *une page de résultats* p_i au client, qui est un

tuple $p_i = \langle \omega_i, S_i \rangle$ où ω_i est l'ensemble des solutions produites durant le quantum q. Le client intelligent est alors libre de continuer l'exécution de sa requête en envoyant S_i au serveur Web.

En conséquence, l'évaluation d'une requête SPARQL avec un serveur Web préemptif créée *une partition des résultats de la requête dans le temps*, car le client Web récupère des ensembles partiels de résultats $\omega_1, \ldots, \omega_n$ durant l'exécution de la requête. Étant donné un jeu de données RDF $\mathcal{D}$ et un serveur Web préemptif avec un quantum de temps q, l'évaluation correcte d'une requête SPARQL Q avec $\mathcal{D}$ en utilisant le serveur produit un ensemble de pages $\mathcal{P} = \{p_1, \ldots, p_n\}$ qui respecte les propriétés suivantes:

1. $[\![Q]\!]_{\mathcal{D}} = \bigcup_{\langle \omega_i, S_i \rangle \in \mathcal{P}} \omega_i$.
2. Il se vérifie que, pour $p_n = \langle \omega_n, S_n \rangle, S_n = nil$.
3. Pour toute paire de deux pages distinctes $p_i = \langle \omega_i, S_i \rangle \in \mathcal{P}, p_j = \langle \omega_j, S_j \rangle \in \mathcal{P}$, il se vérifie que $S_i \neq S_j$.
4. Il existe un ordre total strict $\prec$ sur $\mathcal{P}$ tel que, $\forall p_i = \langle \omega_i, S_i \rangle \in \mathcal{P}, \exists p_j = \langle \omega_j, S_j \rangle \in \mathcal{P}$, où p_j est le successeur direct de p_i selon $\prec$, il se vérifie que $\texttt{Execute}(\texttt{Resume}(S_i)) = \langle \omega_j, x \rangle$ et $\texttt{Suspend}(x) = S_j$.

La première propriété vérifie qu'un client reçoit bien l'ensemble des solutions de sa requête ($[\![Q]\!]_{\mathcal{D}}$) une fois que la dernière page a été produite. La deuxième propriété vérifie que la dernière page produite ne contient aucun état sauvegardé : l'exécution de la requête est terminée et ne peut être reprise. Les deux dernières propriétés vérifient que *l'exécution de la requête progresse au fil du temps* : tous les états sauvegardés sont distincts et produits linéairement dans le temps.

A.3 SaGe: Un moteur préemptif d'évaluation de requêtes SPARQL

Le modèle de la préemption Web permet aux fournisseurs de données de construire des services d'évaluation de requêtes SPARQL qui ne sont pas sensibles aux effets convois et délivrent des résultats complets pour n'importe quelle requête. En revanche, il introduit un surcoût non négligeable durant l'exécution, lors de l'interruption et de la reprise des requêtes. Pour éviter que ce surcoût ne détériore les performances d'exécution, il nous faut le minimiser, *c.a.d.*, minimiser les complexités en temps et en espace des opérations `Suspend` et `Resume`.

Pour ce faire, nous proposons SaGe, un moteur préemptif d'évaluation de requêtes SPARQL qui implémente le modèle de la préemption Web et qui maintient un surcoût minimal. Notre objectif est de borner les complexités citées précédemment de manière à ce qu'elles dépendent uniquement du nombre d'opérations dans un plan d'exécution. En d'autres termes, le problème est de *déterminer*

Itérateur interruptible	Complexité en espace de `Suspend`	Complexité en temps de	
		`Suspend`	`Resume`
Projection $\pi_V(P)$	$\mathcal{O}(\lvert V\rvert + \lvert\, dom(P)\rvert)$ Proposition 1	$\mathcal{O}(\lvert V\rvert + S_P)$ Proposition 2	$\mathcal{O}(R_P)$ Proposition 3
Index Scan $IScan(tp)$	$\mathcal{O}(\lvert tp\rvert + \lvert t\rvert)$ Proposition 4	$\mathcal{O}(\log_b \lvert\mathcal{D}\rvert)$ Proposition 5	$\mathcal{O}(\log_b \lvert\mathcal{D}\rvert)$ Proposition 6
Merge Join $MJoin(P_1, P_2)$	$\mathcal{O}(\lvert\, dom(P_1)\rvert + \lvert\, dom(P_2)\rvert)$ Proposition 7	$\mathcal{O}(S_{P_1} + S_{P_2})$ Proposition 8	$\mathcal{O}(R_{P_1} + R_{P_2})$ Proposition 9
Index Loop Join $IJoin(P, tp)$	$\mathcal{O}(\lvert\, dom(P)\rvert + \lvert tp\rvert + \lvert t\rvert)$ Proposition 10	$\mathcal{O}(S_P + \log_b \lvert\mathcal{D}\rvert)$ Proposition 11	$\mathcal{O}(R_P + \log_b \lvert\mathcal{D}\rvert)$ Proposition 12
Multi-set Union $MUnion(P_1, P_2)$	$\mathcal{O}(1)$ Proposition 13	$\mathcal{O}(S_{P_1} + S_{P_2})$ Proposition 14	$\mathcal{O}(R_{P_1} + R_{P_2})$ Proposition 15
Filter $Filter(P, \mathcal{R})$	$\mathcal{O}(\lvert\, dom(P)\rvert + \lvert\mathcal{R}\rvert)$ Proposition 16	$\mathcal{O}(S_P)$ Proposition 17	$\mathcal{O}(R_P)$ Proposition 18
Physical plan P_Q	$\mathcal{O}(\lvert\, dom(Q)\rvert + \lvert Q\rvert \times \lvert t\rvert)$ Théorème 1	$\mathcal{O}(\lvert Q\rvert \times \log_b \lvert\mathcal{D}\rvert)$ Théorème 2	$\mathcal{O}(\lvert Q\rvert \times \log_b \lvert\mathcal{D}\rvert)$ Théorème 3

Table A.1: Complexités de préemption pour les itérateurs interruptibles.

quels plans d'exécutions P_Q ont un surcoût de préemption borné en $\mathcal{O}(|Q|)$, où $|Q|$ est le nombre d'opérateurs dans le plan P_Q. Nous observons que pour certains opérateurs physiques, leur surcoût de préemption dépasse très largement notre borne de $\mathcal{O}(|Q|)$. Pour résoudre ce problème, nous proposons avec SaGe une série de choix d'implémentation pour le modèle de la préemption Web, afin de miniser le surcoût de préemption [3].

Pour réduire le surcoût de préemption de certains opérateurs physiques, nous choisissons de les répartir entre un serveur SaGe et un client intelligent SaGe. Comme proposé dans l'approche LDF [38, 72], la combinaison du serveur et du client intelligent permet l'exécution n'importe quelle requête SPARQL. La répartition des opérateurs est la suivante :

- Les opérateurs de type "mapping-at-a-time" ont un surcoût relativement faible, ils sont donc supportés par le serveur SaGe. Ces opérateurs forment un sous-ensemble de CoreSPARQL [38], composé des opérateurs Triple Patterns, Basic Graph Pattern `UNION`, `FILTER` et `SELECT`.

- Les opérateurs de type "full-mappings" ont un surcoût trop élevés en moyenne, ils sont donc implementé dans le client intelligent SaGe. Il s'agit des opérateurs `OPTIONAL`, `SERVICE`, `ORDER BY`, `GROUP BY`, `DISTINCT`, `MINUS`, `FILTER EXIST`, ainsi que les opérateurs d'aggrégation (`COUNT`, `AVG`, `SUM`, `MIN`, `MAX`).

[3] Le lecteur notera que l'implémentation proposée dans ce manuscrit est une implémentation possible, mais pas la seule.

La table A.1 résume les complexités en temps et en espace des opérations `Suspend` et `Resume` pour tous les opérateurs. Nous observons que les opérateurs interruptibles Index Scans sont les plus coûteux vis à vis de la préemption Web. De manière générale, les complexité en temps et espace sont supérieures à notre cible initiale en $\mathcal{O}(|Q|)$. Néanmoins, nos résultats expérimentaux (Section 4.4) démontrent qu'en pratique, le surcoût en temps est inférieur à quelques millisecondes et celui en espace à quelques kilobytes.

Comparé aux SPARQL endpoints qui utilisent des politiques de quotas, SaGe évite les effets convois et c'est un pari gagnant dès le moment où les temps d'exécutions des requêtes SPARQL varient significativement. Comparé à des approches LDF [35, 72], SaGe offre une interface serveur plus expressive, avec le support de jointures, unions et filtres évalués côté serveur. En conséquence, il réduit considérablement le transfert de données entre client et serveur, ainsi que le coût de communication, améliorant le temps d'exécution des requêtes SPARQL.

List of Figures

List of Algorithms

List of Tables

Bibliography

[1] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. "SW-Store: a vertically partitioned DBMS for Semantic Web data management". In: *VLDB J.* 18.2 (2009), pp. 385–406. DOI: 10.1007/s00778-008-0125-y.

[2] M. Acosta and M. Vidal. "Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data". In: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*. 2015, pp. 111–127. DOI: 10.1007/978-3-319-25007-6_7.

[3] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. "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*. 2011, pp. 18–34. DOI: 10.1007/978-3-642-25073-6_2.

[4] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. "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. Lecture Notes in Computer Science. Springer, 2014, pp. 197–212. DOI: 10.1007/978-3-319-11964-9_13.

[5] T. Anderson and M. Dahlin. *Operating Systems: Principles and Practice*. 2nd. Recursive books, 2014. ISBN: 0985673524, 9780985673529.

[6] C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche. "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. Lecture Notes in Computer Science. Springer, 2013, pp. 277–293. DOI: 10.1007/978-3-642-41338-4_18.

[7] C. B. Aranda, A. Polleres, and J. 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. Lecture Notes in Computer Science. Springer, 2014, pp. 390–405. DOI: 10.1007/978-3-319-11915-1_25.

[8] R. G. Athreya, A. N. Ngomo, and R. Usbeck. "Enhancing Community Interactions with Data-Driven Chatbots-The DBpedia Chatbot". In: *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*. 2018, pp. 143–146. DOI: 10.1145/3184558.3186964.

[9] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. "DBpedia: A Nucleus for a Web of Open 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*. 2007, pp. 722–735. DOI: 10.1007/978-3-540-76298-0_52.

[10] R. Avnur and J. M. Hellerstein. "Eddies: Continuously Adaptive Query Processing". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. 2000, pp. 261–272. DOI: 10.1145/342009.335420.

[11] T. Berners-Lee. "Linked data-design issues". In: *http://www. w3. org/DesignIssues/LinkedData. html* (2006).

[12] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. "World-wide web: The information universe". In: *Internet Research* 2.1 (1992), pp. 52–58.

[13] T. Berners-Lee, J. Hendler, O. Lassila, et al. "The semantic web". In: *Scientific american* 284.5 (2001), pp. 28–37.

[14] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 0-201-10715-5.

[15] C. Bizer, T. Heath, and T. 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.

[16] M. W. Blasgen, J. Gray, M. F. Mitoma, and T. G. Price. "The Convoy Phenomenon". In: *Operating Systems Review* 13.2 (1979), pp. 20–25. DOI: 10.1145/850657.850659.

[17] D. Comer. "The Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2 (1979), pp. 121–137. DOI: 10.1145/356770.356776.

[18] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll, and B. McBride. "RDF 1.1 concepts and abstract syntax". In: *W3C recommendation* 25.02 (2014). URL: https://www.w3.org/TR/rdf11-concepts/.

[19] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113. DOI: 10.1145/1327452.1327492.

[20] A. Deshpande, Z. G. Ives, and V. Raman. "Adaptive Query Processing". In: *Foundations and Trends in Databases* 1.1 (2007), pp. 1–140. DOI: 10.1561/1900000001.

[21] D. J. DeWitt, J. F. Naughton, and J. Burger. "Nested Loops Revisited". In: *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS 1993), Issues, Architectures, and Algorithms, San Diego, CA, USA, January 20-23, 1993*. 1993, pp. 230–242. DOI: 10.1109/PDIS.1993.253088.

[22] D. Diefenbach, P. H. Migliatti, O. Qawasmeh, V. Lully, K. Singh, and P. Maret. "QAnswer: A Question Answering prototype bridging the gap between a considerable part of the LOD cloud and end-users". In: *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. 2019, pp. 3507–3510. DOI: 10.1145/3308558.3314124.

[23] O. Erling and I. Mikhailov. "Virtuoso: RDF Support in a Native RDBMS". In: *Semantic Web Information Management - A Model-Based Perspective*. 2009, pp. 501–519. DOI: 10.1007/978-3-642-04329-1_21.

[24] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. "Binary RDF representation for publication and exchange (HDT)". In: *J. Web Sem.* 19 (2013), pp. 22–41. DOI: 10.1016/j.websem.2013.01.002.

[25] D. W. Fife. "R68-47 Computer Scheduling Methods and Their Countermeasures". In: *IEEE Trans. Computers* 17.11 (1968), pp. 1098–1099. DOI: 10.1109/TC.1968.226869.

[26] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)* Pearson Education, 2009. ISBN: 978-0-13-187325-4.

[27] G. Graefe. "Query Evaluation Techniques for Large Databases". In: *ACM Comput. Surv.* 25.2 (1993), pp. 73–170. DOI: 10.1145/152610.152611.

[28] G. Graefe and W. J. McKenna. "The Volcano Optimizer Generator: Extensibility and Efficient Search". In: *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*. IEEE Computer Society, 1993, pp. 209–218. DOI: 10.1109/ICDE.1993.344061.

[29] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. "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. URL: http://www.vldb.org/conf/1997/P276.PDF.

[30] S. Harris and N. Gibbins. "3store: Efficient Bulk RDF Storage". In: *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*. 2003.

[31] S. Harris, N. Lamb, N. Shadbolt, et al. "4store: The design and implementation of a clustered RDF store". In: *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*. 2009, pp. 94–109.

[32] S. Harris, A. Seaborne, and E. Prud'hommeaux. "SPARQL 1.1 query language". In: *W3C recommendation* 21.10 (2013), p. 778. URL: https://www.w3.org/TR/sparql11-query/.

[33] A. Harth, K. Hose, and R. Schenkel, eds. *Linked Data Management*. Chapman and Hall/CRC, 2014. ISBN: 978-1-4665-8240-8.

[34] A. Harth, J. Umbrich, A. Hogan, and S. Decker. "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*. 2007, pp. 211–224. DOI: 10.1007/978-3-540-76298-0_16.

[35] O. Hartig and C. B. Aranda. "Bindings-Restricted Triple Pattern Fragments". In: *On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings*. Vol. 10033. Lecture Notes in Computer Science. Springer, 2016, pp. 762–779. DOI: 10.1007/978-3-319-48472-3_48.

[36] O. Hartig, C. Bizer, and J. C. Freytag. "Executing SPARQL Queries over the Web of Linked Data". In: *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*. Vol. 5823. Lecture Notes in Computer Science. Springer, 2009, pp. 293–309. DOI: 10.1007/978-3-642-04930-9_19.

[37] O. Hartig and R. Heese. "The SPARQL Query Graph Model for Query Optimization". In: *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings*. 2007, pp. 564–578. DOI: 10.1007/978-3-540-72667-8_40.

[38] O. Hartig, I. Letter, and J. 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. Lecture Notes in Computer Science. Springer, 2017, pp. 364–382. DOI: 10.1007/978-3-319-68288-4_22.

[39] O. Hartig and J. Pérez. "Semantics and Complexity of GraphQL". In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. 2018, pp. 1155–1164. DOI: 10.1145/3178876.3186014.

[40] R. Hasan. "Predicting query performance and explaining results to assist Linked Data consumption. (Prédire les performances des requêtes et expliquer les résultats pour assister la consommation de données liées)". PhD thesis. University of Nice Sophia Antipolis, France, 2014. URL: https://tel.archives-ouvertes.fr/tel-01127124.

[41] A. Hasnain, Q. Mehmood, S. S. e Zainab, and A. Hogan. "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.

[42] L. Heling, M. Acosta, M. Maleshkova, and Y. Sure-Vetter. "Querying Large Knowledge Graphs over Triple Pattern Fragments: An Empirical Study". In: *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*. 2018, pp. 86–102. DOI: 10.1007/978-3-030-00668-6_6.

[43] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. "MonetDB: Two Decades of Research in Column-oriented Database Architectures". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 40–45.

[44] M. D. Jacyntho and D. Schwabe. "A multigranularity locking model for RDF". In: *J. Web Semant.* 39 (2016), pp. 25–46. DOI: 10.1016/j.websem.2016.05.002.

[45] P. Jesus, C. Baquero, and P. S. Almeida. "A Survey of Distributed Data Aggregation Algorithms". In: *IEEE Communications Surveys and Tutorials* 17.1 (2015), pp. 381–404. DOI: 10.1109/COMST.2014.2354398.

[46] L. Kleinrock. "Analysis of A time-shared processor". In: *Naval research logistics quarterly* 11.1 (1964), pp. 59–73.

[47] E. V. Kostylev, J. L. Reutter, and M. Ugarte. "CONSTRUCT Queries in SPARQL". In: *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*. 2015, pp. 212–229. DOI: 10.4230/LIPIcs.ICDT.2015.212.

[48] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. "RStar: an RDF storage and query system for enterprise resource management". In: *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8-13, 2004*. 2004, pp. 484–491. DOI: 10.1145/1031171.1031264.

[49] F. Maali, I. A. Hassan, and S. Decker. "Scheduling for SPARQL Endpoints". In: *Proceedings of the 10th International Workshop on Scalable Semantic Web Knowledge Base Systems co-located with 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 20, 2014*. 2014, pp. 19–28. URL: http://ceur-ws.org/Vol-1261/SSWS2014_paper2.pdf.

[50] G. Moerkotte and T. Neumann. "Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products". In: *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. 2006, pp. 930–941. ISBN: 1-59593-385-9.

[51] T. Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *PVLDB* 4.9 (2011), pp. 539–550. DOI: 10.14778/2002938.2002940. URL: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[52] T. Neumann and G. 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.

[53] T. Neumann and G. Weikum. "The RDF-3X engine for scalable management of RDF data". In: *VLDB J.* 19.1 (2010), pp. 91–113. DOI: 10.1007/s00778-009-0165-y.

[54] T. Neumann and G. Weikum. "x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases". In: *PVLDB* 3.1 (2010), pp. 256–263. DOI: 10.14778/1920841.1920877.

[55] C. H. Papadimitriou. "The serializability of concurrent database updates". In: *J. ACM* 26.4 (1979), pp. 631–653. DOI: 10.1145/322154.322158.

[56] J. Pérez, M. Arenas, and C. Gutiérrez. "Semantics and complexity of SPARQL". In: *ACM Trans. Database Syst.* 34.3 (2009), 16:1–16:45. DOI: 10.1145/1567274.1567278.

[57] M. Perrin. *Distributed systems: concurrency and consistency*. Elsevier, 2017. ISBN: 9781785482267.

[58] A. Polleres, M. R. Kamdar, J. D. Fernández, T. Tudorache, and M. A. Musen. "A More Decentralized Vision for Linked Data". In: *Proceedings of the 2nd Workshop on Decentralizing the Semantic Web co-located with the 17th International Semantic Web Conference, DeSemWeb@ISWC 2018, Monterey, California, USA, October 8, 2018*. 2018. URL: http://ceur-ws.org/Vol-2165/paper1.pdf.

[59] M. Saleem, Q. Mehmood, and A. N. Ngomo. "FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework". In: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*. 2015, pp. 52–69. DOI: 10.1007/978-3-319-25007-6_4.

[60] M. Schmachtenberg, C. Bizer, and H. 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*. 2014, pp. 245–260. DOI: 10.1007/978-3-319-11964-9_16.

[61] M. Schmachtenberg, C. Bizer, and H. 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. Lecture Notes in Computer Science. Springer, 2014, pp. 245–260. DOI: 10.1007/978-3-319-11964-9_16.

[62] M. Schmidt, M. Meier, and G. Lausen. "Foundations of SPARQL query optimization". In: *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*. ACM, 2010, pp. 4–33. DOI: 10.1145/1804669.1804675.

[63] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. "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. Lecture Notes in Computer Science. Springer, 2011, pp. 601–616. DOI: 10.1007/978-3-642-25073-6_38.

[64] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. "Column-store support for RDF data management: not all swans are white". In: *PVLDB* 1.2 (2008), pp. 1553–1563. DOI: 10.14778/1454159.1454227.

[65] M. Sintek and M. Kiesel. "RDFBroker: A Signature-Based High-Performance RDF Store". In: *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Proceedings*. 2006, pp. 363–377. DOI: 10.1007/11762256_28.

[66] A. Soulet and F. 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*. 2019, pp. 576–592. DOI: 10.1007/978-3-030-30793-6_33.

[67] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. "SPARQL basic graph pattern optimization using selectivity estimation". In: *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*. 2008, pp. 595–604. DOI: 10.1145/1367497.1367578.

[68] R. Taelman, M. V. Sande, and R. Verborgh. "GraphQL-LD: Linked Data Querying with GraphQL". In: *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018*. 2018.

[69] T. P. Tanon, M. D. de Assunção, E. Caron, and F. M. Suchanek. "Demoing Platypus - A Multilingual Question Answering Platform for Wikidata". In: *The Semantic Web: ESWC 2018 Satellite Events - ESWC 2018 Satellite Events, Heraklion, Crete, Greece, June 3-7, 2018, Revised Selected Papers*. 2018, pp. 111–116. DOI: 10.1007/978-3-319-98192-5_21.

[70] A. I. Torre-Bastida, E. Villar-Rodriguez, M. N. Bilbao, and J. D. Ser. "Intelligent SPARQL Endpoints: Optimizing Execution Performance by Automatic Query Relaxation and Queue Scheduling". In: *Algorithms and Architectures for Parallel Processing - 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings*. 2016, pp. 3–17. DOI: 10.1007/978-3-319-49583-5_1.

[71] T. Urhan and M. J. Franklin. "XJoin: A Reactively-Scheduled Pipelined Join Operator". In: *IEEE Data Eng. Bull.* 23.2 (2000), pp. 27–33.

[72] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert. "Triple Pattern Fragments: A low-cost knowledge graph interface for the Web". In: *J. Web Sem.* 37-38 (2016), pp. 184–206. DOI: 10.1016/j.websem.2016.03.003.

[73] D. Vrandecic and M. Krötzsch. "Wikidata: a free collaborative knowledgebase". In: *Commun. ACM* 57.10 (2014), pp. 78–85. DOI: 10.1145/2629489.

[74] H. Wagner, C. Golden, and S. R. Bissette. *Prince of Stories: The Many Worlds of Neil Gaiman*. St. Martin's Press, 2008.

[75] C. Weiss, P. Karras, and A. Bernstein. "Hexastore: sextuple indexing for semantic web data management". In: *PVLDB* 1.1 (2008), pp. 1008–1019. DOI: 10.14778/1453856.1453965.

[76] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. "Efficient RDF Storage and Retrieval in Jena2". In: *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8, 2003*. 2003, pp. 131–150.

[77] A. N. Wilschut and P. M. G. Apers. "Dataflow Query Execution in a Parallel Main-memory Environment". In: *Distributed and Parallel Databases* 1.1 (1993), pp. 103–128. DOI: 10.1007/BF01277522.

[78] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. "MonetDB/X100 - A DBMS In The CPU Cache". In: *IEEE Data Eng. Bull.* 28.2 (2005), pp. 17–22.

Acronyms

AST Abstract Syntax Tree

FCFS First-Come First-Served scheduling policy

HDT Header, Dictionary, Triples compression format

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IRI International Resource Identifier

LDF Linked Data Fragments

LDFM Linked Data Fragments Machine

LOD Linked Open Data

RDF Resource Description Framework

RR Round Robin scheduling policy

SPARQL Simple Protocol And RDF Query language

URI Unique Resource Identifier

URL Uniform Resource Locator

W3C The World Wide Web Consortium

WatDiv Waterloo SPARQL Diversity Test Suite

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

La Préemption Web pour interroger le Web des Données

Mots-clés : Web sémantique · Gestion des données liées · Serveurs SPARQL publics

Résumé : en suivant les principes du Linked Open Data, les fournisseurs de données ont publié des milliards de documents RDF via des services publics d'évaluation de requêtes SPARQL. Pour garantir la disponibilité et la stabilité de ces services, ils appliquent des politiques de quotas sur l'utilisation des serveurs. Les requêtes qui excèdent ces quotas sont interrompues et ne renvoient que des résultats partiels. Cette interruption n'est pas un problème s'il est possible de reprendre l'exécution des requêtes ultérieurement, mais il n'existe aucun modèle de préemption le permettant. Dans cette thèse, nous proposons de résoudre le problème relatif à la construction des services qui permettent à n'importe quel utilisateur d'exécuter n'importe quelle requête SPARQL en obtenant des résultats complets. Nous proposons *la préemption Web*, un nouveau modèle d'exécution qui permet l'interruption de requêtes SPARQL après un quantum de temps, ainsi que leur reprise sur demande des clients. Nous proposons également SaGe, un moteur d'évaluation de requêtes SPARQL qui implémente la préemption Web tout en garantissant un surcoût de préemption minimal. Nos résultats expérimentaux démontrent que SaGe est plus performant que les approches existantes, en termes de temps moyen d'exécution des requêtes et d'obtention des premiers résultats.

Web Preemption for Querying the Linked Open Data

Keywords : Semantic Web · Linked Data Management · Public SPARQL servers

Abstract: Following the Linked Open Data principles, data providers have published billions of RDF documents using public SPARQL query services. To ensure these services remains stable and responsive, they enforce quotas on server usage. Queries which exceed these quotas are interrupted and deliver partial results. Such interruption is not an issue if it is possible to resume queries execution afterward. Unfortunately, there is no preemption model for the Web that allows for suspending and resuming SPARQL queries. In this thesis, we propose to tackle the issue of building public SPARQL query servers that allow any data consumer to execute any SPARQL query with complete results. First, we propose a new query execution model called *Web Preemption*. It allows SPARQL queries to be suspended by the Web server after a fixed time quantum and resumed upon client request. Web preemption is tractable only if its cost in time is negligible compared to the time quantum. Thus, we propose SaGe: a SPARQL query engine that implements Web Preemption with minimal overhead. Experimental results demonstrate that SaGe outperforms existing SPARQL query processing approaches by several orders of magnitude in term of the average total query execution time and the time for first results.