On the Efficient Distributed Evaluation of SPARQL Queries

Damien Graux

Université Grenoble Alpes

ON THE EFFICIENT DISTRIBUTED EVALUATION OF SPARQL QUERIES

Damien Graux

Supervisor: Nabil Layaïda
Co-Supervisor: Pierre Genevès

– Jury Members –
Rapporteur: Mohand-Saïd Hacid
Rapporteur: Patrick Valduriez
Examinateur: Jérôme Euzenat
Examinateur: Farouk Toumani
Supervisor: Nabil Layaïda
Co-Supervisor: Pierre Genevès

December, 15th 2016

Foretaste

> The Semantic Web,
> On Distributed SPARQL.
> PhD. Thesis

The Semantic Web standardized by the World Wide Web Consortium aims at providing a common framework that allows data to be shared and analyzed across applications. The Resource Description Framework (RDF) and the query language SPARQL constitute two major components of this vision.

Because of the increasing amounts of RDF data available, dataset distribution across clusters is poised to become a standard storage method. As a consequence, efficient and distributed SPARQL evaluators are needed.

To tackle these needs, we first benchmark several state-of-the-art distributed SPARQL evaluators while monitoring a set of metrics which is appropriate in a distributed context (*e.g.* network traffic). Then, an analysis driven by typical use cases leads us to define new development perspectives in the field of distributed SPARQL evaluation. On the basis of these perspectives, we design several efficient distributed SPARQL evaluators whose performances are validated and compared to state-of-the-art evaluators. For instance, our distributed SPARQL evaluator named SPARQLGX[1] offers efficient time performances while being resilient to the loss of nodes.

[1] http://github.com/tyrex-team/sparqlgx

Abstract

Context. The Semantic Web aims at providing a common framework that allows data to be shared and reused across application. The increasing amounts of RDF data available raise a major need and research interest in building efficient and scalable distributed SPARQL query evaluators.

Contributions. In this work, in order to constitute a common basis of comparative analysis, we first evaluate on the same cluster of machines various SPARQL evaluation systems from the state-of-the-art. These experiments lead us to point several observations: (i) the solutions have very different behaviors; (ii) most of the benchmarks only consider temporal metrics and forget other ones *e.g.* network traffic. That is why, we propose a larger set of metrics; and thanks to a reading grid based on 5 criteria, we propose new perspectives when developing distributed SPARQL evaluators:

1. *Velocity*: applications might favour the fastest possible answers.
2. *Immediacy*: applications might need to be ready as fast as possible to only evaluate some SPARQL queries only once.
3. *Dynamicity*: applications might need to deal with dynamic data.
4. *Parsimony*: applications might need to execute queries while minimizing some of the cluster resources.
5. *Resiliency*: applications might tolerate the loss of machines while processing.

Then, we develop and share several distributed SPARQL evaluators which take into account these new considerations we introduced:

– A SPARQL evaluator named SPARQLGX[1]: an implementation of a distributed RDF datastore based on Apache Spark. SPARQLGX is designed to leverage existing Hadoop infrastructures for evaluating SPARQL queries. SPARQLGX relies on a translation of SPARQL queries into executable Spark code that adopts evaluation strategies according to (1) the storage method used and (2) statistics on data. We show that SPARQLGX makes it possible to evaluate SPARQL queries on billions of triples distributed across multiple nodes, while providing attractive performance figures.

– Two SPARQL direct evaluators *i.e.* without a preprocessing phase:

- SDE[1] (stands for **S**PARQLGX **D**irect **E**valuator): it lays on the same strategy than SPARQLGX but the translation process is modified in order to take the orign data files as argument.
- RDFHive[2]: it evaluates translated SPARQL queries on-top of Apche Hive which is a distributed relational data warehouse based on Apache Hadoop.

[1]https://github.com/tyrex-team/sparqlgx
[2]https://github.com/tyrex-team/rdfhive

Related Publications.

SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark.
Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda.
The 15th International Semantic Web Conference, Oct 2016, Kobe, Japan.

SPARQLGX in action: Efficient Distributed Evaluation of SPARQL with Apache Spark.
Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda.
The 15th International Semantic Web Conference, Oct 2016, Kobe, Japan.

Smart Trip Alternatives for the Curious.
Damien Graux, Pierre Genevès, Nabil Layaïda.
The 15th International Semantic Web Conference, Oct 2016, Kobe, Japan.

Résumé

Contexte. Le Web Sémantique est une extension du Web standardisée par le *World Wide Web Consortium*. Les différents standards utilisent comme format de base pour les données le *Resource Description Framework* (RDF) et son langage de requêtes nommé SPARQL. Plus généralement, le Web Sémantique tend à orienter l'évolution du Web pour permettre de trouver et de traiter l'information plus facilement. Ainsi, avec l'augmentation des volumes de données RDF disponibles, de nombreux efforts de recherche ont été faits pour permettre l'évaluation distribuée et efficace de requêtes SPARQL ; en effet, il s'avère que les solutions de stockage de données distribuées sont de plus en plus répandues.

Contributions. Dans cette étude, afin de constituer une base commune d'analyse comparative, nous commençons par évaluer au sein d'un même ensemble de machines plusieurs solutions d'évaluation de requêtes SPARQL issues de l'état-de-l'art. Ces expérimentations nous permettent de mettre en évidence plusieurs points : (i) tout d'abord, les comportements des différentes solutions observées sont très variables ; (ii) la plupart des méthodes de comparaison ne semblent pas permettre de comparer efficacement les solutions dans un contexte distribué puisqu'elles se cantonnent à des considérations temporelles et négligent l'utilisation des ressources *e.g.* les communications réseau. C'est pourquoi, afin de mieux observer les comportements, nous avons étendu l'ensemble des métriques considérées et proposé de nouvelles problématiques dans la conception de tels évaluateurs ; en effet, nous considérons les critères suivants pour notre nouvelle grille de lecture :

1. La *vélocité* où l'application doit répondre le plus rapidement possible aux requêtes.
2. L'*immédiateté* où l'application doit être prête à évaluer des requêtes au plus vite.
3. La *dynamicité* où les données peuvent évoluer dans le temps.
4. La *parcimonie* où l'utilisation des ressources disponibles doit être mesurée.
5. La *robustesse* où les évaluateurs doivent être capables de supporter la chute de machines au sein de l'ensemble des nœuds.

Puis, toujours dans le contexte distribué, nous proposons et partageons différents évaluateurs SPARQL en tenant compte de ces nouvelles considérations que nous avons développés :

– Un évaluateur SPARQL nommé SPARQLGX[1] : une solution de stockage RDF distribuée basée sur Apache Spark et utilisant les infrastructures Hadoop pour évaluer des requêtes SPARQL. SPARQLGX repose sur un traducteur de requêtes SPARQL vers une séquence d'instructions exécutables par Spark en adoptant des stratégies d'évaluation selon (1) le schéma de stockage des données utilisé et (2) des statistiques sur les données. Nous montrons que SPARQLGX permet l'évaluation de requêtes SPARQL sur plusieurs milliards de triplets RDF répartis sur plusieurs nœuds, tout en ayant des performances attractives.

[1] https://github.com/tyrex-team/sparqlgx

– Deux évaluateurs SPARQL directs *i.e.* sans phase de chargement préalable :

  - SDE[1] (acronyme de **S**PARQLGX **D**irect **E**valuator) : repose sur la méthode de traduction déjà mise en place pour SPARQLGX adaptée afin d'évaluer directement la requête sur les données RDF originelles.
  - RDFHive[2] : permet d'évaluer des requêtes SPARQL grâce à Apache Hive qui est une infrastructure de stockage de données distribuées interrogeable via un langage relationel.

Publications Associées.

SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark.
Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda.
The 15th International Semantic Web Conference, Oct 2016, Kobe, Japan.

SPARQLGX in action: Efficient Distributed Evaluation of SPARQL with Apache Spark.
Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda.
The 15th International Semantic Web Conference, Oct 2016, Kobe, Japan.

Smart Trip Alternatives for the Curious.
Damien Graux, Pierre Genevès, Nabil Layaïda.
The 15th International Semantic Web Conference, Oct 2016, Kobe, Japan.

[2]https://github.com/tyrex-team/rdfhive

Remerciements

Avant toute chose, je tiens à adresser mes remerciements aux responsables : Nabil Layaïda et Pierre Genevès qui m'ont permis d'intégrer l'équipe Tyrex sous les meilleurs auspices !

En outre, je tiens aussi à remercier tout particulièrement les membres du jury ayant accepté d'évaluer mon travail *id est* mes rapporteurs : Messieurs Patrick Valduriez et Mohand-Saïd Hacid ainsi que mes examinateurs : Messieurs Farouk Toumani et Jérôme Euzenat.

Ensuite, je remercie l'ensemble des membres de l'équipe Tyrex (passés et présents) avec lesquels il est toujours agréable de travailler.

Enfin, c'est à ma famille et à mes proches que je souhaite adresser mon éternelle gratitude pour l'indéfectible soutien dont ils ont toujours su faire preuve.

Merci à tous ! ☺

Contents

List of Figures

List of Tables

Introduction

We first introduce the context in which this thesis takes place and some required background elements. The scientific problem addressed is then presented. We then recap the set of contributions we realized. Finally, we present the organization of the present dissertation.

Background & Context

The Semantic Web. The general context is the *Semantic Web* which refers to a Web of data that is readable and processable by machines. Initially, Tim Berners-Lee coined this term [19] in 2001:

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

However, such a notion has already been outlined by Tim Berners-Lee in the first World Wide Web Conference (W3C) in 1994 and later in Weaving the Web [54].

The main purpose of the Semantic Web is to allow both machines and humans to treat and manage data that can be found in the Web; and then they should be able to infer meaning (and knowledge) from these information in order to assist users in their activities. To achieve these goals, several steps of standardization are required; W3C working groups have thereby written recommandations and standards to describe the multiple subparts the Semantic Web is composed of.

The Semantic Web is often informally represented using a layer stack initially created by Tim Berners-Lee, depicted in its forth version [18] in Figure {i} and described in particular in [54]. This representation illustrates the basic core concepts of the Semantic Web; in addition, it provides a hierarchy since each layer uses capabilities of the ones below. This stack can be divided into three blocks. First, at the bottom, layers contain well-known technologies used for hypertext Web and which constitute a basis for the Semantic Web *i.e.* URI [66], Unicode [29] and XML [23]. The W3C standards requiered to build Semantic Web applications constitute the middle layers: RDF [53], SPARQL [72], RDFS [24], OWL [35] and RIF [58]. Finally, the top layers group technologies that are not currently standardized: Cryptography, "Trust"-layers and user interface.

Currently Published Standards. Up to now, several standards of the stack illustrated in Figure {i} have already been developed by the W3C. We present them in a nutshell starting from the bottom with the standards that are part of the hypertext Web:

1. Uniform Resource Identifiers (URIs) [66] have been standardized by the IETF in 2005. A URI is a string of characters which provides a mean to uniquely identify a Semantic Web resource.

2. Unicode [29] serves to represent and manipulate text in many languages.

3. The Extensible Markup Language (XML) [23] defines a set of rules for encoding document. One of the Semantic Web goal is to give meaning to structured documents.

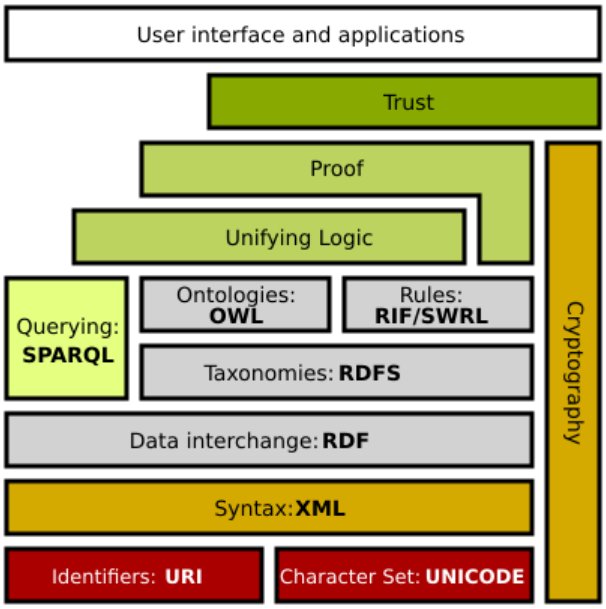


Figure {i}: Semantic Web Layer Cake.

As explained previously, the middle layers of the Semantic Web stack represent the core concepts of the Semantic Web.

4. The Resource Description Framework (RDF) [53] is a framework for creating statements called triples and enables to represent information about resources in the form of graphs.

5. SPARQL [72] is a RDF query language.

6. RDF Schema (RDFS) [24] is a set of classes having properties and using the RDF data model which can provide basic description of ontologies. Such classes are called RDF Vacabularies.

7. The Web Ontology Language (OWL) [35] is an extension of RDFS allowing more advanced constructs to describe RDF data.

8. The Rule Interchange Format (RIF) [58] is used to exchange rules between the various specific "rules languages" of the Semantic Web.

More generally, a detailed description of the Semantic Web stack and its variants can be found in the study of Gerber *et al.* in [42].

Focus & Problem

In the context of this thesis, we mainly focus on two W3C standards of the Semantic Web: the RDF and SPARQL. Indeed, the increasing amounts of RDF data available raise a major need and research interest in building efficient SPARQL query evaluators where RDF datasets are distributed across storage/computation nodes because of their size. That is why, we center our study in the case of distributed SPARQL evaluation. This choice is also motivated by the advent of inexpensive distributed hardware which implies the possibility of setting up clusters more easily. For these reasons, we consider that dataset distribution across clusters is poised to become a standard storage method. The general problem is to provide the community new designs for RDF storage and efficient new SPARQL evaluators while considering the distributed context by adapting/extending the traditional methods used to rank such systems.

Summary of Contributions

First of all, to constitute a common basis of comparative analysis, we evaluate on the same cluster of machines various SPARQL evaluation systems from the literature. These experiments lead us to point several observations: (i) the solutions have very different behaviors; (ii) most of the benchmarks only use temporal metrics and forget other ones *e.g.* network traffic. That is why, we propose a larger set of metrics; and thanks to a new reading grid based on 5 features, we propose new perspectives which should be considered when developing distributed SPARQL evaluators.
Second, we develop and share several distributed SPARQL evaluators which take into account these new considerations we introduced. (1) A SPARQL evaluator named SPARQLGX[1]: an implementation of a distributed RDF datastore based on Apache Spark. SPARQLGX is designed to leverage existing Hadoop infrastructures for evaluating SPARQL queries. SPARQLGX relies on a translation of SPARQL queries into executable Spark code that adopts evaluation strategies according to the storage method used and statistics on data. We show that SPARQLGX makes it possible to evaluate SPARQL queries on billions of triples distributed across multiple nodes, while providing attractive performance figures. (2) Two SPARQL direct evaluators *i.e.* without a preprocessing phase: SDE[1] (stands for **S**PARQLGX **D**irect **E**valuator) lays on the same strategy than SPARQLGX but the translation process is modified in order to take the orign data files as argument. RDFHive[2] evaluates translated SPARQL queries on-top of Apache Hive which is a distributed relational data warehouse based on Apache Hadoop.

Thesis Outline

The rest of the dissertation is divided as follows into two main parts. The first one reviews the current state-of-the-art and the needed tools and notions for the rest of the development. The second part focuses on the efficient SPARQL evaluation in a cluster-based context. More specifically, we subdivise the document into the following chapters:

Part I
– Chapter 1 starts with an introduction to first Semantic Web layer: RDF. We describe its concepts, models and specifications and review some of its most popular syntaxes.
– Chapter 2 presents in detail the RDF query language called SPARQL.
– Chapter 3 is twofold. First it lists the various technics that can be used to store RDF datasets and how such methods have been set up to develop centralized RDF datastores. Second, it focuses on state-of-the-art strategies used to distribute on several nodes the evaluation of SPARQL queries on multi-nodes RDF datasets.

Part II
– Chapter 4 is an experimental work we realized on various state-of-the-art distributed SPARQL evaluators we selected in order to have a common basis of comparison for the solutions we developed.
– Chapter 5 presents in details SPARQLGX: its architecture, its technical steps and its performance compared with previously benchmarked systems.
– Chapter 6 is an extension of the Chapter 4. Indeed, considering the already studied systems it extends the set of metrics to fit a distributed context; and thereby it offers new perspectives in the development of distributed RDF/SPARQL-based applications based on a multi-criteria approach.
– Chapter 7 considers the particular case of direct SPARQL evaluation introduced in Chapter 6 and presents in detail two SPARQL evaluators we developed: RDFHive and SDE.

[1]Sources: `https://github.com/tyrex-team/sparqlgx`
[2]Sources: `https://github.com/tyrex-team/rdfhive`

– Chapter 8 is the presentation of a practical example of application where heterogeneous sources of different sizes need to be merged and/or queried sequentially. We describe in this Chapter an enriched trip planner.

Part I

State-of-the-art in Distributed SPARQL Evaluation

Table of Contents

In this Part, we present the general context of this study. To do so, we start by presenting the two W3C standards we focus on: Chapter 1 introduces RDF and Chapter 2 describes SPARQL. Then, we review the existing RDF storage methods and various SPARQL evaluator strategies in a distributed context in Chapter 3.

Chapter 1

Resource Description Framework

The Resource Description Framework (RDF) is a language standardized by W3C to express structured information on the Web as graphs [53]. It models knowledge about arbitrary resources using Unique Resource Identifiers (URIs), Blank Nodes and Literals. RDF data is structured in sentences – or triples – written (s p o), each one having a subject s, a predicate p and an object o. This Chapter introduces this language and some of its concepts. More specifically, we will first describe the goals of such a framework in Section 1.1. Then in Section 1.2 we introduce its concept using example before describing RDF model in Section 1.3. Finally, we review some popular RDF syntaxes in Section 1.6.

Contents

The Resource Description Framework (RDF) is a language for representing information about resources in the Web, in particularly to represent metadata about resources (*e.g.* titles, authors, ...). In addition, this "Web resource" content can be extended to describe information about things that can be identified on the Web, such as descriptions of Web authors or relationships between people.

By construction, RDF is more intended for situations in which data need to be processed automatically, rather than being only displayed to people since its formalism is not "user-friendly". Moreover, the framework is designed to share and distribute easily data between applications and sources.

Basically, RDF designates things and concepts using *Uniform Resource Identifiers* (URIs) and describes relations between them in terms of simple properties.

The current RDF specification [53] is split into six W3C Recommendations. The most important document is the RDF Primer, which introduces the basic concepts of RDF. It is a summary of the other documents and contains the basic information needed to effectively use RDF. The RDF Primer also describes how to define vocabularies using the RDF Vocabulary Description Language (also called RDF Schema).

1.1 RDF Motivations & Goals

First of all, the development of RDF has been motivated by practical uses:

- *Sharing*: Web resources and systems using them may want to propagate information about themselves such as content descriptions, capability descriptions, privacy preferences, authors, creation or modification dates...
- *Flexibility*: Applications may prefer opening information models rather than constrained ones to easily share data with other applications such as schedules, processes...
- *Independence*: The authors may allow their data to be processed outside the environment they created it.
- *Interworking*: Some application may want to aggregate and combine data from various sources to build new pieces of information
- *Lingua Franca*: Software agents may process such information automatically and thus build a world-wide network where processes cooperate directly instead of having to understand human-readable contents.

Therfore, RDF is designed to represent information in a minimally constraining and flexible way. Indeed, it can be used in isolated applications or directly between several ones or even in various projects that do not share the same initial goals. The value of information improves as RDF data are accessible across the Internet.

More particularly, the resource description framework has been designed to reach the following goals:

- presenting a simple way of representing data in order to be easily used by applications.
- having a formal semantics to offer basis for reasoning; more particularly supporting rules of inference in RDF data.
- having an extensible vocabulary thanks to the use of uniform resource identifiers which can be used to name all kinds of things in RDF.
- providing an XML-based syntax to encode, share and exchange datasets between applications.
- allowing anyone to make statements about any resource.

1.2 RDF Concepts

As briefly presented before, the Resource Description Framework aims at representing statements about Web resources. Thus, to introduce basic ideas, we should try to state a simple fact, for instance let's imagine that someone named John developed a software. Such a fact can be explained in English with the following sentence:

> **Example 1:**
> The **software 'foo'** has a **creator** who is **John**.

We notice that three elements of our example-sentence are important (emphasized here), indeed:

- the element the sentence describes (the software 'foo')
- the specific property which is concerned in the sentence (creator)

- the element that is the property value (John)

Using the same type of sentences, we can also improve the description of 'foo':

Example 2:
The **software 'foo'** has a **language** which is **English**.
The **software 'foo'** has a **creation year** which is **2016**.

RDF uses as starting point the postulate that the things it describes have properties which have values and thus that RDF statements can be constructed using similar structures as those presented in the previous basic examples. In the same time, RDF introduces a specific terminology to designate the various parts of its statements: the *subject* is the thing the statement is about, the *predicate* refers to the property and the *object* is its value.

To be widely used, RDF only needs to be machine-processable *i.e.* (1) elements of sentences have to be identified without confusion (unicity of representation) and (2) statements have to be represented by an easily machine-processable language/syntax.
On the one hand, the Web already provides a way to identify pages with *Uniform Resource Locators* (URLs) which are strings representing access mechanisms (usually through network); however, it seems important to also reference things and concepts that do not have "webpages" and thus no URLs. That is why, RDF proposes to use a more general identifying concept: *Uniform Resource Identifiers* (URIs) which incorporates URLs. URIs have the property of not being limited to network-reachable concept and thus can refer to anything *e.g.* actors, trees in streets, abstract concepts...
On the other hand, to make statements machine-processable, various languages and syntaxes (see Section 1.6) can be standardized, for instance RDF borrows markup language elements such as "<" or ">" to materialize URIs.

1.3 A URI-based RDF Model

As explained in Section 1.2, RDF uses URI references (URIrefs) for identification and machine-processable representations (*e.g.* XML markup elements) in its syntax. We will in this Section describe how the Resource Description Framework allows to make "semantic" statements. Since each statement consists of triple composed of a subject, a predicate, and an object, the Example 1 (Section 1.2) could be represented by:

- subject: http://www.software-foo.org/home.html
- predicate: http://purl.org/dc/elements/1.1/creator
- object: http://www.software-foo.org/id/01

We can note here that URIrefs are used to identify each component of the statement: even the object "John" which could have also been represented by a simple string of characters. (Such a use of URIrefs will be explained later.)

In the same time, we can also see an RDF statement according to a graph representation. Indeed, a statement can be seen in terms of nodes and arcs by:

- a node for the subject
- a node for the object
- an arc for the predicate which is directed from the subject node to the object one

So the RDF statement above would be represented by the graph shown in Figure 1.1. Similarly (see Figure 1.2), the join of Examples 1 & 2 can be represented by a group nodes and arcs.

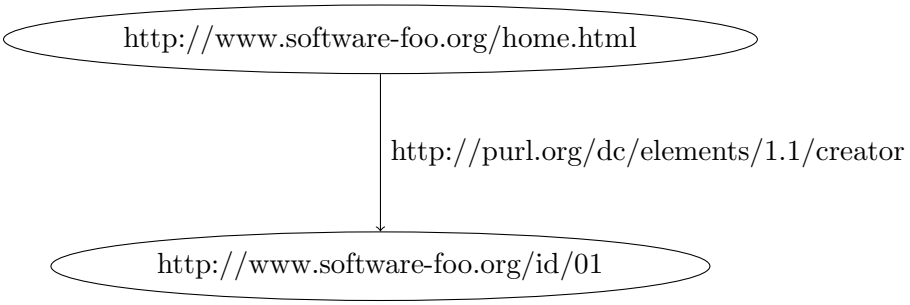


Figure 1.1: Graph of the Example 1

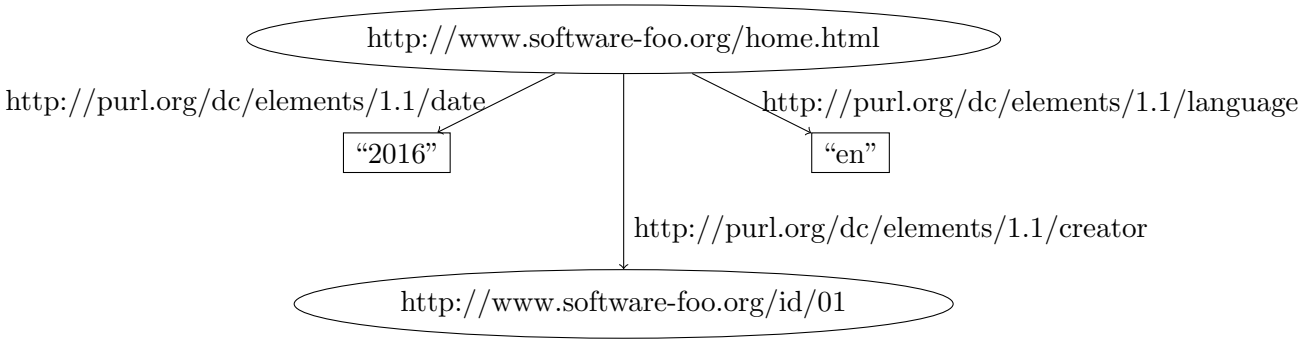


Figure 1.2: Graph Examples 1 & 2

We show in Figure 1.2 that objects may be either URIrefs or *literals i.e.* constant values. Theses character strings are strictly reserved for objects in RDF statements. In Figure 1.2, the literals are *plain* which signifies that they are not typed.

Alternatively, instead of drawing a graph, one can represent RDF statements writting *triples*. In other words, each graph statement is written as a simple triple of subject, predicate and object; each triple represents a single arc in the graph. Considering the three triples considered in Examples 1 & 2, we have:

Example 3:

```
<http://www.software-foo.org/home.html> <http://purl.org/dc/elements/1.1/creator> <http://www.software-foo.org/id/01>
<http://www.software-foo.org/home.html> <http://purl.org/dc/elements/1.1/language> "en"
<http://www.software-foo.org/home.html> <http://purl.org/dc/elements/1.1/date> "2016"
```

The “full triples” notation requires complete URIrefs written in angle brackets (*i.e.* “<” and “>”); this requirement (see Example 3) results in long lines. For convenience, the RDF Primer introduces a shorthand way of writing triples using prefixes to substitute name without brackets as abbreviation for a full URIref. For instance, we can clean the Example 3 using two prefixes as above; and more generally, RDF Primer also introduces several “well-known” prefixes defined in Table 1.1.

Example 4:

```
prefix foo: <http://www.software-foo.org/>
prefix dc: <http://purl.org/dc/elements/1.1/>
foo:home.html dc:creator foo:id/01
foo:home.html dc:language "en"
foo:home.html dc:date "2016"
```

Practically, a prefix can also be used to define vocabularies. For instance, one can organize URIrefs according to common URI prefixes. However, it is just a way of vizualisation since the RDF

prefix	namespace
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
dc:	http://purl.org/dc/elements/1.1/
owl:	http://www.w3.org/2002/07/owl#
xsd:	http://www.w3.org/2001/XMLSchema#

Table 1.1: Common RDF prefixes

model does not extract knowledge reading URIrefs. In the same time, vocabularies can be "mixed" using for instance URIrefs of various sources in the same desciptions or statements. Moreover, RDF allows to reuse *ad libitum* a same URIref in a graph. For example, if the software "foo" has been co-developed by several people, several statements will represent this concept:

```
foo:home.html dc:creator foo:id/01
foo:home.html dc:contributor foo:id/42
foo:home.html dc:contributor foo:id/02
...
```

More generally, using as much URIrefs as possible for objects provides several advantages. First, it is more precise than a literal; second since it is a resource, it allows the records of additional information.
Moreover, RDF only uses URIrefs for predicates to identify properties which important for several reasons. It sets the meaning of the considered property. In addition, using URIrefs to identify properties enables the properties to be treated as resources themselves; since properties are resources, additional information can be recorded about them.

> " Using URIrefs as subjects, predicates, and objects in RDF statements supports the development and use of shared vocabularies on the Web, since people can discover and begin using vocabularies already used by others to describe things, reflecting a shared understanding of those concepts. "
>
> RDF Primer [53]

However, URIrefs do not solve all the identification and interpretation problems because people can still use different references to describe the same thing. That is why using terms coming from existing vocabularies can be a good behavior. Nonetheless, the result of all this is that RDF provides a way to make statements that applications can more easily process.

1.4 Existential Variables: Blank Nodes

Unfortunately, recorded information are not always in the form of the simple RDF statements illustrated so far. Indeed, real-world data involves complicated structures. For instance, let's consider the INRIA postal address written out as a plain literal:

```
prefix ex: <http://example.org/inria/>
ex:lab ex:address "655 Av de l'Europe, 38330 Montbonnot-Saint-Martin, France" .
```

Now, depending on the use case, we might need this address recorded as a structure of seperate number, street, city, postal code...to do that in RDF, we will need to introduce intermediate triples. Structured information like this is represented in RDF by considering the aggregate thing to be described as a separate resource, and then making statements about that new resource.

So, in the graph, in order to break up INRIA's address into its component parts, a new node is created to represent the concept of INRIA's address, with a new URIref to identify it, say RDF

statements *i.e.* additional arcs and nodes can then be written with that node as the subject, to represent the additional information. This way of representing structured information in RDF can involve generating numerous "intermediate" URIrefs. Since a complex graph might contain more than one blank node, we need to be able to distinguish them. As a result, blank nodes have the following form _:name. For instance, in this example a blank node identifier _:inriaAddress might be used to refer to the blank node, in which case the resulting triples might be:

```
prefix ex: <http://example.org/inria/>
ex:lab           ex:address         _:inriaAddress .
_:inriaAddress   ex:number          "655" .
_:inriaAddress   ex:street          "Av de l'Europe" .
_:inriaAddress   ex:city            "Montbonnot-Saint-Martin" .
_:inriaAddress   ex:state           "France" .
_:inriaAddress   ex:postalCode      "38330" .
```

Because blank node identifiers represent (blank) nodes, rather than arcs, in the triple form of an RDF graph, blank node identifiers may only appear as subjects or objects in triples and never as predicates.

1.5 Typed Literals

Lastly, sometimes plain literal parts have to be structured. For instance, to diplay a date, one will need to explode it into month, day, number. However, so far, constant values that represent objects in RDF statements are plain (untyped) literals – even when the value is probably a number *e.g.* an age, a year... Usually, the common practice in programming languages is to append additional information about the interpretation of a literal with the literal.

We thus construct an RDF typed literal by appending a string with a URIref that corresponds to a datatype. For instance, John's age could be described as being the integer number "31" using the triple:

```
foo:id/01 foo:age "27"^^xsd:integer
```

Similarly, in our example, the value of the page's dc:date property is written as the plain literal "2016". However, using a typed literal, it could be explicitly described as being the date September, 15th 2016 with:

```
foo:home.html dc:date "2016-09-15"^^xsd:date
```

Unfotunately, unlike several programming languages, RDF has no pre-defined datatype sets: it simply provides a way to indicate how a literal could be interpreted.

1.6 Common RDF Syntaxes

As described in the previous Section 1.3, RDF has a graph conceptual model; in addition, we saw that the triple notation is a shortway of representing RDF data. In this Section, we will introduce some popular RDF syntaxes.

1.6.1 N-Triples

The N-Triples RDF syntax [3] has been motivated by the will of building simple RDF parsers. Indeed, the N-Triple standard is a line-based RDF syntax, in other words, each triple is fully written on a line. As a consequence, each RDF element of a triple – subject, predicate and object – should be

written without any kind of abbreviation such as prefixes. The RDF fields are then separated with either spaces or tabulations and each statement always finishes with a dot.

More specifically, URIrefs are written between angle brackets *i.e.* "<" and ">", blank nodes are preceded by "_:" and literal are enclosed by double quotes, they can be language-tagged using a "@" and typed with "^^". For instance, we present after four RDF triples expressed according to the N-Triples format:

```
<http://example/ntriple/subj> <http://example/pred1> "object"@en .
<http://example/ntriple/subj> <http://example/pred2> _:obj .
_:obj <http://example/property1> <http://example/something> .
_:obj <http://example/property2> "A short text here." .
```

1.6.2 Turtle

Turtle [27] is a textual syntax for RDF which defines itself as a "terse" method to write RDF graphs with a compact form. Somehow, Turtle can be seen as a extension of the N-Triples representation introduced previously: indeed, a simple triple statement in Turtle is a sequence of subject, predicate and object separated by spaces or tabulations and ended by a dot like in N-Triples.

Blank Nodes. They are written using the syntax as the one introduced in Section 1.4 *e.g.* "_:name".

URIs. In Turtle, URIs have to be enclosed between angle brackets < and >. In addition, they can also be written relatively since Turtle allows shortcuts to be more human-readable: bases and prefixes can be defined to deal with long *uri*s. For instance:

```
# Original URI:
<http://example.org/very/longUri/1>
# With a base:
@base <http://example.org/very/longUri/> .
<1>
# With a prefix:
@prefix pref: <http://example.org/very/longUri/> .
pref:1
```

Literals. Plain literals are written between double quotes in Turtle. Additionally, literals can be typed appending a URI after two circumflexes. Moreover, the language-tag can also be specified with the at sign "@".

```
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema/> .
ex:book1 ex:authorName "Mr Doe" .
ex:book1 ex:title "Un livre"@fr .
ex:book1 ex:title "A book"@en .
ex:book1 ex:date "2016-09-15"^^xsd:date .
```

Predicate & Object Lists. Finally, Turtle provides ways to compact RDF statements using two facts: (1) the same subject is often referenced by several predicates and (2) the same (subject,predicate) couple can have number of objects. In both cases, Turtle allows to make lists, actually ";" is used to repeat the subject of triples that vary only in predicates and objects and "," has the same role to repeat (subject,predicate) couple differing only in objects. For instance, using the group of triples previously lists can be compacted according to:

```
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema/> .
ex:book1 ex:title "A book" .
ex:book1 ex:date "2016-09-15"^^xsd:date .
ex:book1 ex:authorName ex:id/01 .
```

(a) RDF Triples written in Turtle.

```
01.<?xml version="1.0"?>
02.<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
03.         xmlns:ex="http://example.org/">
04.  <rdf:Description rdf:about="http://www.example.org/book1">
05.    <ex:title>A book</ex:title>
06.  </rdf:Description>
07.  <rdf:Description rdf:about="http://www.example.org/book1">
08.    <ex:date rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
          2016-09-15
       </ex:date>
09.  </rdf:Description>
10.  <rdf:Description rdf:about="http://www.example.org/book1">
11.    <ex:authorName rdf:resource="http://example.org/id/01"/>
12.  </rdf:Description>
13.</rdf:RDF>
```

(b) The RDF/XML Translation.

Figure 1.3: RDF/XML Basic Example.

```
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema/> .
ex:book1 ex:authorName "Mr Doe" ;
         ex:date "2016-09-15"^^xsd:date ;
         ex:title "Un livre"@fr ,
                  "A book"@en .
```

1.6.3 RDF/XML

RDF provides an XML syntax for writing down RDF graphs, called RDF/XML. Unlike triples, which are intended as a shorthand notation, RDF/XML is the normative syntax for writing RDF introduced in [17].

To briefly present RDF/XML, we will use as an illustration the correspondance between RDF triples in Turtle in Figure 1.3a and the translation in terms of RDF/XML in Figure 1.3b which will be described line by line:
Line 1 – It is the XML declaration and its version.
Lines 2 & 3 – It starts an `rdf:RDF` element. On the same line, an XML namespace (`xmlns`) is done; similarly, an other one is declared on Line 3 before closing the tag with the angle bracket.
Lines 4 to 6 – These three lines represent a triple using `rdf:Description` where the subject is in the `about` field Line 4. The predicate is declared on Line 5 in its own tag. The plain literal is then written between the two predicate tags.
Lines 7 to 9 – Like the previous group, they define an RDF triple, however, we can notice here (on Line 8) that the object is a typed literal; this specification is done thanks to the `datatype` field.
Lines 10 to 12 – They deal with the case of a resource objet (*i.e.* a URIref). In such a case, the

element is written using an XML empty-element tag (no separate end-tag), and the property value is written using an `rdf:resource` attribute within that empty element.
Line 13 – Finally, this last line clauses the `rdf:RDF` declarations.

1.7 RDF Essentials

As introduced in this Chapter, RDF is a simple model, consisting of the following fundamentals:

- An RDF graph is a set of RDF triples
- An RDF triple has three components:
  - an RDF subject, which is an RDF URI reference or a blank RDF node
  - an RDF predicate, which is an RDF URI reference
  - an RDF object, which is an RDF URI reference, a blank RDF node or an RDF literal
- An RDF literal can be of two kinds:
  - an RDF plain literal is a character string with an optional associated language tag describing the language of the character string
  - an RDF typed literal is a character string with an associated RDF datatype URI. An RDF datatype defines the syntax and semantics of a set of character strings that represent data such as booleans, integers, dates, etc.

Chapter 2

SPARQL

SPARQL is the standard query language for retrieving and manipulating data represented in the Resource Description Framework (RDF) [53]. SPARQL constitutes one key technology of the semantic web and has become very popular since it became an official W3C recommendation [72, 46].

Contents

In the Semantic Web, querying RDF data is mainly realized using the **S**PARQL **P**rotocol and **R**DF **Q**uery **L**anguage *i.e.* SPARQL. It became a standard thanks to the RDF data access working group (DAWG). On January 15th 2008, SPARQL1.0 [72] becomes an official W3C recommandation, before being updated into SPARQL1.1 [46] in March 2013.

In this Chapter, we present the foundations of SPARQL and its syntax. To do so, we first introduce few required concepts in Section 2.1. Then, we introduce the generic anatomy of a SPARQL query in Section 2.2. Sections 2.3, 2.4, 2.5, 2.6 & 2.7 present more in details the various elements of a SPARQL query: respectively the headers, the clauses, the different possible forms, the solution modifiers and the selection of the dataset. Finally, we briefly review previous studies dealing with complexity of SPARQL evaluation in Section 2.8.

2.1 Common Definitions

This Section describes formal concepts that will be used all along this work. More partcularly, we first present concepts already introduced in the Chapter 1 that are required to provide the ones related to this Chapter.

First of all, we introduce two concepts directly related to the RDF standard dealing with the various sets of entities possible in RDF statements.

RDF Term. Remembering the Section 1.7, we first formalize the following sets:

- Let I be the set of all URIs.
- Let L be the set of all RDF Literals.
- Let B be the set of all blank nodes in RDF graphs.

Therefore, the set of RDF *Terms*, RDF_T, is:

$$\text{RDF}_T = I \cup L \cup B$$

RDF Dataset. And more generally, an RDF dataset is a set:

$$\{G, (< u_1 >, G_1), (< u_2 >, G_2), \ldots, (< u_n >, G_n)\}$$

where G and each G_i are graphs, and each $< u_i >$ is a URI. Each $< u_i >$ is distinct. G is called the *default graph*. $(< u_i >, G_i)$ are called *named graphs*.

We can now introduce more SPARQL-specific concepts.

Active Graph. The *active graph* is the graph from the dataset used for basic graph pattern matching.

Query Variables. A *query variable* is a member of the set V where V is infinite and disjoint from RDF_T.

Triple Patterns. A *triple pattern* is member of the set: $(\text{RDF}_T \cup V) \times (I \cup V) \times (\text{RDF}_T \cup V)$ We can notice that this official definition of triple pattern includes literal subjects whereas in the Section 1.7 we forbad that according to the RDF. That is why we will adapt the formal definition of triple patterns:

$$(I \cup B \cup V) \times (I \cup V) \times (\text{RDF}_T \cup V)$$

Basic Graph Patterns. A *Basic Graph Pattern* is a set of Triple Patterns *i.e.* a conjunction of triple patterns. The empty graph pattern is a basic graph pattern which is the empty set.

Solution Mapping. A *solution mapping* is a mapping from a set of variables to a set of RDF terms. We use the term "solution" where it is clear. More formally, a solution mapping, μ, is a partial function $\mu : V \to T$. The domain of μ, namely $dom(\mu)$, is the subset of V where μ is defined.

Solution Sequence. A *solution sequence* is a list of solutions, possibly unordered.

SPARQLQuery	:=	[Header*] Form [Dataset] `WHERE` { Pattern } Modifiers
Header	:=	`PREFIX` value value | `BASE` value
Form	:=	`SELECT` [`DISTINCT`|`REDUCED`] (joker|var*) | `ASK` | `CONSTRUCT` var* | `DESCRIBE`
Dataset	:=	`FROM` value | `FROM Named` value
Modifiers	:=	`LIMIT` value | `OFFSET` value | `ORDER By` [`ASK`|`DESC`] var*
Pattern	:=	Pattern . Pattern | {Pattern} `UNION` {Pattern} | Pattern `OPTIONAL` {Pattern}
	|	(value|var) (value|var) (value|var) | `FILTER` Constraint
var	:=	(‘?’|‘$’)value
joker	:=	‘*’
value	∈	$\mathbb{S}$tring

Figure 2.1: Basic Syntax of a SPARQL Query.

2.2 Anatomy of a SPARQL query

SPARQL has a SQL-like syntax. In Figure 2.1, we present a simplified version of the full SPARQL syntax for the purposes of our study; we thus refer the reader to the official W3C recommendation in [46] for more details.

Mainly, a SPARQL query can be divided into five parts (Figure 2.1). First, optional headers can be given in order to make the rest of the query more human-readable. Second, the standard allows several query forms that modify the shape of the results. Third, optional clauses on the RDF sources can be set to specify the dataset against which the query is executed. Then, the `WHERE` clause, which is the core of SPARQL query, specifies in its terms a set of conditions used to compose the result. Finally, optional solution modifiers, operating over the triples selected by the `WHERE` clauses, can be set to refine the selection before generating the results.

In the rest of this Chapter we will explain deeper the behavior of each components before briefly concluding with common definitions.

2.3 Query Headers

SPARQL allows writing-shortcuts to make query more human-readable using the same syntax as the Turtle one, already introduced in Section 1.6.2. A optional base and an optional list of prefixes can be defined on-top of the query. These declarations are useful to deal with long URIrefs. For instance, here are two variations around an original URI:

```
# Original URI:
<http://example.org/very/longUri/1>
# With a base:
BASE <http://example.org/very/longUri/>
<1>
# With a prefix:
PREFIX pref: <http://example.org/very/longUri/>
pref:1
```

2.4 Query Clauses

SPARQL is based around graph pattern matching specified in the `WHERE` clause. More complex graph patterns can be formed by combining smaller patterns in various ways:

- Basic Graph Patterns, where a set of triple patterns must match
- Group Graph Pattern, where a set of graph patterns must all match

- Optional Graph patterns, where additional patterns may extend the solution
- Alternative Graph Pattern, where two or more possible patterns are tried
- Patterns on Named Graphs, where patterns are matched against named graphs

2.4.1 Conjunctivity

It exists two forms that combine patterns by conjunction: *basic graph patterns* which combine triples patterns and *group graph patterns* which combine all other graph patterns.

Basic graph patterns are conjunctive sets of triple patterns see Section 2.1 for a definition of triple pattern. SPARQL graph pattern matching is defined in terms of combining the results from matching basic graph patterns.

In a SPARQL query, a group graph pattern is delimited with braces: {}. For example, this query's query pattern is a group graph pattern of one basic graph pattern.

```
SELECT ?name ?mbox
WHERE  {  ?x foaf:name ?name .
          ?x foaf:mbox ?mbox .
       }
```

The same solutions would be obtained from a query that grouped the triple patterns into two basic graph patterns. For example, the query below has a different structure but would yield the same solutions as the previous query:

```
SELECT ?name ?mbox
WHERE  { { ?x foaf:name ?name . }
         { ?x foaf:mbox ?mbox . }
       }
```

The group pattern:{} matches any graph (including the empty graph) with one solution that does not bind any variables. For example: `SELECT ?x WHERE {}` matches with one solution in which variable x is not bound.

A constraint, expressed by the keyword `FILTER`, is a restriction on solutions over the whole group in which the filter appears.

2.4.2 Potentiality

Basic graph patterns allow applications to make queries where the entire query pattern must match for there to be a solution. For every solution of a query containing only group graph patterns with at least one basic graph pattern, every variable is bound to an RDF Term in a solution. However, regular, complete structures cannot be assumed in all RDF graphs. It is useful to be able to have queries that allow information to be added to the solution where the information is available, but do not reject the solution because some part of the query pattern does not match. Optional matching provides this facility: if the optional part does not match, it creates no bindings but does not eliminate the solution.

Optional parts of the graph pattern may be specified syntactically with the `OPTIONAL` keyword applied to a graph pattern:

```
pattern OPTIONAL { pattern }
```

The syntactic form:

```
{ OPTIONAL { pattern } }
```

is equivalent to:

```
{ { } OPTIONAL { pattern } }
```

The `OPTIONAL` keyword is left-associative :

```
pattern OPTIONAL { pattern } OPTIONAL { pattern }
```

is the same as:

```
{ pattern OPTIONAL { pattern } } OPTIONAL { pattern }
```

In an optional match, either the optional graph pattern matches a graph, thereby defining and adding bindings to one or more solutions, or it leaves a solution unchanged without adding any additional bindings.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .
@prefix rdf:        <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
_:a  rdf:type        foaf:Person .
_:a  foaf:name       "Alice" .
_:a  foaf:mbox       <mailto:alice@example.com> .
_:a  foaf:mbox       <mailto:alice@work.example> .
_:b  rdf:type        foaf:Person .
_:b  foaf:name       "Bob" .

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x  foaf:mbox  ?mbox }
       }

name mbox
"Alice" <mailto:alice@example.com>
"Alice" <mailto:alice@work.example>
"Bob"
```

We notice that there is no value of mbox in the solution where the name is “Bob”. In details, this query finds the names of people in the data. If there is a triple with predicate mbox and the same subject, a solution will contain the object of that triple as well. In this example, only a single triple pattern is given in the optional match part of the query but, in general, the optional part may be any graph pattern. The entire optional graph pattern must match for the optional graph pattern to affect the query solution.

Graph patterns are defined recursively. A graph pattern may have zero or more optional graph patterns, and any part of a query pattern may have an optional part. In this example, there are two optional graph patterns.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .
_:a  foaf:name       "Alice" .
_:a  foaf:homepage   <http://work.example.org/alice/> .
_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@work.example> .

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox ?hpage
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x foaf:mbox ?mbox } .
```

```
        OPTIONAL { ?x foaf:homepage ?hpage }
      }
```

```
name mbox hpage
"Alice" <http://work.example.org/alice/>
"Bob" <mailto:bob@work.example>
```

2.4.3 Alternativity

SPARQL provides a means of combining graph patterns so that one of several alternative graph patterns may match. If more than one of the alternatives matches, all the possible pattern solutions are found.

Pattern alternatives are syntactically specified with the `UNION` keyword.

```
@prefix dc10:  <http://purl.org/dc/elements/1.0/> .
@prefix dc11:  <http://purl.org/dc/elements/1.1/> .
_:a  dc10:title     "Book1" .
_:a  dc10:creator   "Alice" .
_:b  dc11:title     "Book2" .
_:b  dc11:creator   "Bob" .
_:c  dc10:title     "Book3" .
_:c  dc11:title     "Book3.1" .
```

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>
SELECT ?title
WHERE  {
 { ?book dc10:title  ?title }
 UNION
 { ?book dc11:title  ?title }
}
```

```
title
"Book2"
"Book3"
"Book3.1"
"Book1"
```

This query finds titles of the books in the data, whether the title is recorded using Dublin Core properties from version 1.0 or version 1.1. To determine exactly how the information was recorded, a query could use different variables for the two alternatives:

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>
SELECT ?x ?y
WHERE  { { ?book dc10:title ?x } UNION { ?book dc11:title  ?y } }
```

```
x | y
  | "SPARQL (updated)"
  | "SPARQL Protocol Tutorial"
"SPARQL"  |
"SPARQL Query Language Tutorial"  |
```

This will return results with the variable x bound for solutions from the left branch of the `UNION`, and y bound for the solutions from the right branch. If neither part of the `UNION` pattern matched, then the graph pattern would not match. The `UNION` pattern combines graph patterns; each alternative possibility can contain more than one triple pattern.

2.5 Query Forms

SPARQL probides four query forms. These query forms use the solutions from pattern matching to form result sets or RDF graphs. The query forms are the following ones:

- `SELECT` which returns all, or a subset of, the variables bound in a query pattern match.
- `CONSTRUCT` which returns an RDF graph constructed by substituting variables in a set of triple templates.
- `ASK` which returns either "*yes*" or "*no*" indicating if a query pattern matches or not.
- `DESCRIBE` which returns an RDF graph that describes the resources found.

2.5.1 SELECT

The `SELECT` form of results returns variables and their bindings directly. The syntax "`SELECT *`" is an abbreviation that selects all of the variables in a query.

```
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .
_:a    foaf:name   "Alice" .
_:a    foaf:knows  _:b .
_:a    foaf:knows  _:c .
_:b    foaf:name   "Bob" .
_:c    foaf:name   "Clare" .
_:c    foaf:nick   "CT" .

PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?nameX ?nameY ?nickY
WHERE
  { ?x foaf:knows ?y ;
       foaf:name ?nameX .
    ?y foaf:name ?nameY .
    OPTIONAL { ?y foaf:nick ?nickY }
  }

nameX nameY nickY
"Alice" "Bob"
"Alice" "Clare" "CT"
```

2.5.2 CONSTRUCT

The `CONSTRUCT` query form returns a single RDF graph specified by a graph template in the `WHERE`. The result is an RDF graph formed by taking each query solution in the solution sequence, substituting for the variables in the graph template, and combining the triples into a single RDF graph by set union.

If any such instantiation produces triples containing unbound variables or an illegal RDF statements (such as a literal in subject or predicate position) then these triples are not included in the

final output RDF graph. A particular case is the following: if the graph template (in the WHERE clauses) contains triples with no variables which also appear in the queried dataset, then these triples are part of the returned RDF graph.

```
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .
_:a    foaf:name   "Alice" .
_:a    foaf:mbox   <mailto:alice@example.org> .

PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT   { <http://example.org/person#Alice> vcard:FN ?name }
WHERE       { ?x foaf:name ?name }

@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
<http://example.org/person#Alice> vcard:FN "Alice" .
```

2.5.3 ASK

Applications can use the ASK form to test whether or not a query pattern has a solution. The query then just returns "*yes*" or "*no*". In particular, no information is returned about the possible query solutions.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .
_:a  foaf:name       "Alice" .
_:a  foaf:homepage   <http://work.example.org/alice/> .
_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@work.example> .

PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
ASK  { ?x foaf:name  "Alice" }

yes

PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
ASK  { ?x foaf:name  "Alice" ;
          foaf:mbox  <mailto:alice@work.example> }

no
```

2.5.4 DESCRIBE (Informative)

The DESCRIBE form returns a single result RDF graph containing RDF data about resources. This data is not prescribed by a SPARQL query, where the query client would need to know the structure of the RDF in the data source, but, instead, is determined by the SPARQL query processor.

The query pattern (in WHERE) is used to create a result set. The DESCRIBE form takes each of the resources identified in a solution, together with any resources directly named by IRI, and assembles a single RDF graph by taking a "description" which can come from any information available including the target RDF dataset. The description is determined by the query service. The syntax "DESCRIBE *" is an abbreviation that describes all of the variables in a query.

2.6 Query Modifiers

Query patterns generate unordered collections of solutions, each solution being a partial function from variables to RDF terms. These solutions are then treated as a sequence (a solution sequence), initially in no specific order. Sequence modifiers can then be applied to create another sequence. A solution sequence modifier is one of:

- Order modifier: puts the solutions in order
- Projection modifier: chooses only some distinguished variables
- Distinct modifier: ensures solutions in the sequence are unique
- Reduced modifier: allows elimination of some non-unique solutions
- Offset modifier: controls where the solutions start from in the overall sequence of solutions
- Limit modifier: restricts the number of solutions

Modifiers are applied in the order given by the list above.

2.6.1 Ordering Solution

The `ORDER BY` clause establishes the order of a solution sequence considering a sequence of order comparators, composed of an expression and an optional order modifier (either `ASC()` or `DESC()`).

```
PREFIX ex:    <http://example.org/>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE { ?x foaf:name ?name ; ex:Id ?num }
ORDER BY ?name DESC(?num)
```

Using `ORDER BY` on a solution sequence for a `CONSTRUCT` or `DESCRIBE` query has no direct effect because only `SELECT` returns a sequence of results. Used in combination with `LIMIT` and `OFFSET`, `ORDER BY` can be used to return results generated from a different slice of the solution sequence. An `ASK` query does not include `ORDER BY`, `LIMIT` or `OFFSET`.

2.6.2 Projection.

The solution sequence can be transformed involving only a subset of the variables. For each solution in the sequence, a new solution is formed using a specified selection of the variables using the `SELECT` query form.

2.6.3 Duplicate Solutions.

A solution sequence with no `DISTINCT` or `REDUCED` query modifier will preserve duplicate solutions. For instance:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:x    foaf:name    "Alice" .
_:y    foaf:name    "Alice" .
_:z    foaf:name    "Alice" .

PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?name WHERE { ?x foaf:name ?name }
```

```
name
"Alice"
"Alice"
"Alice"
```

The DISTINCT solution modifier eliminates duplicate solutions. Specifically, each solution that binds the same variables to the same RDF terms as another solution is eliminated from the solution set. While the DISTINCT modifier ensures that duplicate solutions are eliminated from the solution set, REDUCED simply permits them to be eliminated. The cardinality of any set of variable bindings in a REDUCED solution set is at least one and not more than the cardinality of the solution set with no DISTINCT or REDUCED modifier. Using the last example to continue:

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?name WHERE { ?x foaf:name ?name }

name
"Alice"

PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT REDUCED ?name WHERE { ?x foaf:name ?name }
```

This last query may have one, two or three solutions.

2.6.4 Offset

OFFSET simply causes the solutions generated to start after the specified number of solutions. As a consequence, an OFFSET of zero has no effect.

2.6.5 Limitation

LIMIT puts an upper bound on the number of solutions returned. If the number of actual solutions is greater than the limit, then at most the limit number of solutions will be returned.

2.7 Query Dataset

As described in Chapter 1, the RDF data model expresses information as graphs consisting of triples with subject, predicate and object. As a consequence, many RDF datastores hold multiple RDF graphs and record information about each graph, allowing an application to make queries that involve information from more than one graph. For these reasons, a SPARQL query may specify the dataset to be used for matching by using the FROM clause and the FROM NAMED clause to select the wanted RDF dataset. These two keywords allow a query to specify an RDF dataset by reference; they indicate that the dataset should include graphs that are obtained from representations of the resources identified by the given URIs. The dataset resulting from a number of FROM and FROM NAMED clauses is:

- A default graph consisting of the RDF merge of the graphs referred to in the FROM clauses,

and

- A set of (URI, graph) pairs, one from each FROM NAMED clause.

If there is no FROM clause, but there is one or more FROM NAMED clauses, then the dataset includes an empty graph for the default graph. Each FROM clause contains an URI that indicates a graph to be used to form the default graph. This does not put the graph in as a named graph. In the following example, the RDF dataset contains a single default graph and no named graphs:

```
# Default graph (stored at http://example.org/from/aliceFoaf)
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .
_:a  foaf:name     "Alice" .
_:a  foaf:mbox     <mailto:alice@work.example> .

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT  ?name
FROM    <http://example.org/from/aliceFoaf>
WHERE   { ?x foaf:name ?name }

name
"Alice"
```

2.8 Survey on Complexity of SPARQL Evaluation

A fundamental issue in every query language is the complexity of query evaluation and the influence of each component of the language in this complexity. We briefly discuss here several fragments of SPARQL built incrementally and present complexity results for each such fragment. Indeed, according to the set of allowed SPARQL keywords used – the considered fragment –, the evaluation complexity may differ a lot.

As it is customary when studying the complexity of the evaluation problem for a query language [84], we consider its associated decision problem. We denote this problem by evaluation and we define it as follows:

- *Input*: An RDF graph G, a graph pattern P and a mappinng μ.
- *Output*: Does $\mu \in [\![P]\!]_G$?

1. Starting with the graph pattern expression constructed by using only the operators "AND" – *i.e.* the Basic Graph Pattern fragment introduced in Section 2.1 –, the complexity of evaluation is $O(|P| \times |G|)$ [71], where$|G|$ and $|P|$ are the size of G and P.

2. The complexity of evaluation rises to NP-Complete [71], when the BGP fragment is extended with the `UNION` operator.

3. The complexity of the full SPARQL is PSPACE-Complete [71]. The `OPTIONAL` operator was identified as one of the main sources of complexity. Indeed, it was shown in [80] that the PSPACE-Completeness of SPARQL query evaluation holds even if we restrict SPARQL to the BGP fragment extended with the `OPTIONAL` operator.

In the rest of this study, we will mainly focus on the BGP fragment since it represents the core of SPARQL. Moreover, we will deal with SPARQL query in a distributed context and explain how to develop efficient evaluation strategies.

Chapter 3

RDF Storage Methods & SPARQL Evaluators

This Chapter focuses on the current solutions developed to efficiently store and query RDF data using SPARQL. It is mainly twofold: we first present the single node systems and common RDF storage methods; and second, after presenting popular tools for distributed computing, we review the distributed datastores whose development is more recent.

Contents

We previously introduced two W3C standards: the Resource Description Framework (RDF) in Chapter 1 to represent Semantic data and its associated query language SPARQL in Chapter 2. In the present Chapter, we focus on describing the various methods available to store RDF datasets and query them using SPARQL.

More specifically, we first review the most popular methods to store RDF data on a single machine while presenting centralized stores using these strategies in Section 3.1. Then, after the introduction of specific and popular cloud-based frameworks used by RDF management systems in Section 3.2, we describe the various strategies existing in distributed RDF management in Section 3.3. Finally, we list some already realized surveys and review the popular benchmarks available to compare SPARQL evaluators in Section 3.4.

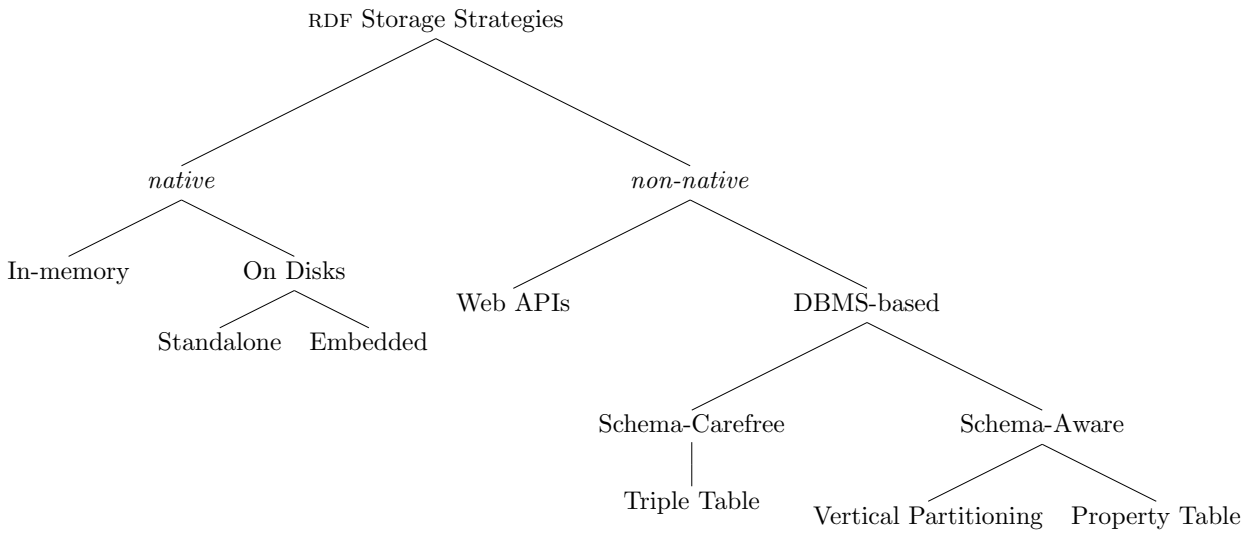


Figure 3.1: Taxonomy of common single-node RDF storage strategies [39].

3.1 RDF Storage Methods

In this Section, we focus on developed strategies to store RDF datasets on single machines; we will further discuss how these methods can be adapted in the case of distributed datastores. Additionally, we also present several stores for each described storage method. A storage has to offer both scalability and performance to be efficient. With the increasing amounts of available RDF datasets, various strategies to store them have emerged. More generally, the literature on RDF storage systems can be divided into two large sets: first the *native* solutions which are RDF compliant and the *non-native* ones. Figure 3.1 shows the considered taxonomy of RDF storage approaches in the context of single-node systems. We indeed further divide the native and non-native sets into smaller blocks we will develop next.

3.1.1 Non-Native Storage Approaches

Mainly, the *non-native* solutions use other database systems previously developed for an other purpose. Actually, they adapt them to the case of storing RDF data. This strategy implies an adaptation of RDF statements to make them fit with the original required format. In Figure 3.1, we divide non-native systems in two groups. The first one mainly encompasses native web applications where RDF data is for instance obtain from XML documents and from HTML pages. The second group encompasses storage strategies relying on relational database management systems.

Conceptually, the *relational model* organizes data into tables of columns and rows. Rows are also called records or tuples. Generally, each table/relation represents one "entity type". The rows represent instances of that type of entity and the columns representing values attributed to that instance.
Currently, efficient storage has already been discussed in the literature – see *e.g.* [16, 39] – with different physical organization methods namely *triple table*, *vertical partitioning* and *property table*. The choice of a strategy rather of than another has impacts on query performances and on dataset modification performances.
In the rest of this section, we review the set of methods to store RDF data in relational systems and give in each case examples of popular stores already developed.

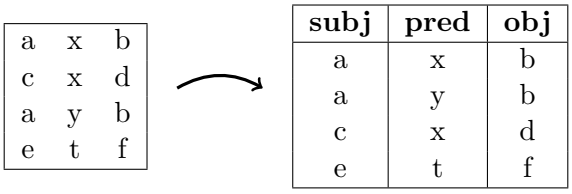


Figure 3.2: Triple Table Model.

Triple Table

The *Triple Table* strategy is probably the most straightforward mapping of RDF into a relational database management system. It actually only considers one single table to store a whole dataset *i.e.* each RDF statement (*s p o*) is stored as one single row composed of three fields respectively subject, predicate and object (see *e.g.* Figure 3.2). In addition, in some approaches this storage model is improved with indexes added for each of the columns in order to make joins less expensive during computations. With this model, updating the database becomes straightforward since a piece of data is stored once in a unique table. However, because of this single table (potentially very large) table, several limitations can appear:

- At scale – depending of the available RAM, the underneath relational system may crash if the whole single table cannot fit in memory.
- At query – queries may be slow to execute; indeed, evaluating complex queries involving multiple triple patterns requires several self-joins over the single table as pointed out in *e.g* [86, 59, 85].

This table approach has been used by systems like: 3store [50], Redland [14], rdfDB [47] or RDF-Store [75].

Property Table

In order to improve RDF storage allowing multiple triple pattern referencing the same subject to be retrieved without expensive joins, the *property table* method has been set up. In this model, each named table includes a subject and several fixed predicates. The concept is to discover clusters of subjects which often appear with the same set of predicates. A popular variant of this scheme is the *property-class table* where the widely used `rdf:type` (see Chapter 1) predicate is used to cluster similar set of subjects together in the same table. As a consequence, self-joins on the subject column can be avoided.

However, in the same time, the model suffers from several drawbacks:

- generating many `NULL` values since, for a given cluster, not all properties will be defined for all subjects mainly due to the fact that RDF is semi-structured.
- it is hard to express multi-value attributes: seeking for all defined predicates of a given subject will imply to scan all tables.
- adding predicates implies to add new tables.

More generally, the property table storage approach loses the flexibility offered by a standard such as RDF. Moreover, queries with triple patterns involving multiple property tables are still expensive because they may require several union clauses and joins to combine data from these tables. This statement complicates the translation of SPARQL queries and the generation of an efficient join plan. For instnce, this approach has been adopted by: Sesame [25], Jena2 [86] or RDFSuite [6].

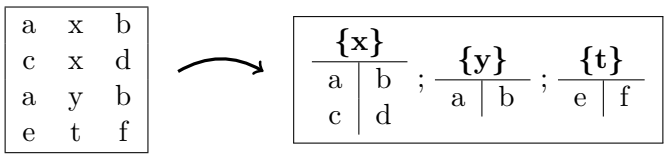


Figure 3.3: Vertical Partitioning Model.

Vertical Partitioning

Abadi *et al.* in [4] present an alternative to the property table called the *vertical partitioning* which tends to improve the query evaluation performances while being easier to implement. In this case, each dataset is divided into k independant tables where k is the number of distinct predicates. Each of these tables is then composed of two columns: the first one contains the subject and the second one the object (see *e.g.* Figure 3.3). To validate their model, Abadi *et al.* also proposed a system implementing the vertical partitioning called swStore in [5].

Even if such an approach may suffer from RDF datasets having a large number of distinct predicates, the vertical partitioning has advantages:

- It easily supports *subject-subject* joins if tables are sorted by subjects. It is then possible to use *merge joins* to reconstruct information about multiple properties for subsets of subjects.
- It provides a support for multi-valued attributes.

3.1.2 Native Storage Approaches

We group RDF storage methods which are closed to data triple model under the *native* denomination. More precisely, we subdivise these solutions (see Figure 3.1) into two subsets: first the one saving them on disks and second the one keeping datasets in-memory.

Firstly, the persistent storages save the needed information permanently on disks using for instance indexes, or structures such as B-Trees. Among these solutions, we can distinguish two types. On the one hand, some are standalone and their unique goal is dedicated to RDF storage using for instance standard representations introduced in Chapter 1 such as RDF/XML or N-Triples... On the other hand, embedded RDF representations are part of specific applications. Obviously, searching across disks may have impact of query performances, however, it provides a form of resiliency in case of a machine reboot.

Lastly, the in-memory systems allocate an amount of the computer main memory to store the whole RDF graph. They can use in details same storage strategies as the disk-based systems do; however, the pre-processing steps might be computed when the system is reboot which represent generally some of the most time-consuming operations for such systems.

Technically, native RDF solutions try to get rid of classic relational system drawbacks such as the definition of inflexible schemas. They thus provide methods to *rearrange* RDF data so that query evaluations can be more performant compared with straightforward storage methods such as the triple table (which is popular among the non-native solutions). More particularly, they often rely on multi-index constructions. Actually, this approach can maintain up to six indexes to cover all the possible schemes in RDF that SPARQL queries may require. Namely, these are PSO, POS, SPO, SOP, OPS and OSP where S,P,O stand respectively for Subject, Predicate and Object; they materialize all the possible orders of precedence of the three RDF elements. In the rest of this subsection, we briefly present some native RDF stores using indexing strategies which have been developed in the last decade:

RDF-3X. In [68], Neumann & Weikum introduced RDF-3X. It uses indexes for all the possible permutations of subject, predicate and object. Parallely, they build their own storage implementation to store the possibly large triple table. Technically, all triples are first lexicagraphically ordered and second stored in comrpessed B+-trees.

YARS. Yet Another RDF Store (YARS) describes optimized index structures to store RDF in [52]. First of all, instead of storing triples, it saves quads where the fourth element is the context which corresponds to the origin of the triple. Element of quads are then encoded in a dictionary mapping literal and URIs to identifiers. In addition, it stores on disks the datasets using six B+-tree indexes whose key is a concatenation of s,p,o,c and which cover all the possible access patterns.

Virtuoso. Like YARS, Virtuoso [38] – which is a commercial system – also stores quads. However, instead of storing the origin of the triple in a context like YARS, it stores the graph. Technically, it saves quads using a single four-column table. In addition, two indexes are computed: *g,s,p,o* and *o,g,p,s* ones; and in the same time, URIs are encoded with a dictionary. Thus, this slight number of indexes tends to focus more on insertion performances rather than on the query performances of Virtuoso.

Hexastore. An approach which adopt both vertical partitioning and multi-indexing has been presented in [85] named Hexastore. As its named suggests, it relies on the construction of six indexes held in-memeory. Additionally, Weiss *et al.* explained that in SPO and PSO the values for O are the same; therefore, five copies of the whole dataset are computed to build six indexes. In order to limit the size of the allocated memory, Hexastore also encodes URIs and literals in a dictionary. With such a strategy, Hexastore favors query performances instead of loading performances since updates or insertions have impacts on the six indexes.

BitMat. Atre *et al.* present in [11] an in-memory bit-matrix structure to store RDF where its multi-join algorithm ensures that intermediate results remain small while no cartesian product is needed. Conceptually, an RDF triple can be seen as a 3-dimensional element; thereby, BitMat has been developed to look like a 3-dimensional bit-cube where each cell represents a unique triple. Practically, this bit-cube is flattened in a 2-dimentional bit-matrix and thus it exists six different ways to realize this step. Moreover, BitMat offers optimizations to efficiently deal with the sparsity of this cube.

Parliament. Kolas *et al.* introduced an RDF native store in [59] which is based on linked lists and memory files. In details, their storage structure is threefold: a resource table, a statement table and a resource dictionary. The first table is a single table of records, each of which represents a single resource or literal. This allows direct access to a record given its ID via simple array indexing. The second one is the most important of Parliament since it encodes RDF statements. Finally, the dictionary is used to reduce the size of the other tables.

3.2 Multi-Node Management Tools

Since the number of available open datasets (RDF ones in particular) and their respective size are both increasing, a possible strategy to scale is to distribute these datasets across clusters. Indeed, the velocity of data generation paired with the richness of data structure motivated researchers to develop computing tools on top of parallel shared-nothing architecture of machines. *Itaque*, various industries shared their own structures of Big Data analysis: Google's MapReduce first in 2004 and

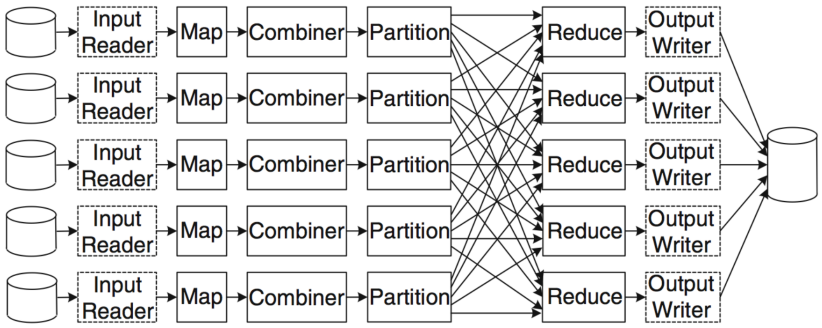


Figure 3.4: MapReduce Dataflow [37].

next in 2008 [34], Microsoft's SCOPE [88], Yahoo's PNUTS [30], LinkedIn's Kafka [44], Walmart-Labs' Muppet [62], Twitter's Storm [64]... In this Section, we present available and popular tools of the literature used to achieve such a distribution.

3.2.1 The MapReduce Paradigm

Among the various tools presented in the beginning of this Section, the MapReduce [34] paradigm developed at first by Google is poised to become one of the most used concepts to distribute computations across a cluster of machines.

More specifically, MapReduce [34] is a framework for parallel processing of massive datasets. Conceptually, a *job* to be performed has to be constructed as two separate phases: first a Map function specifies the *map* phase which takes key/value pairs as input, performs (if necessary) computations and returns key/value pairs as output; second a Reduce function specifies the *reduce* phase where key/value pairs from the Map are ingested to return a single set of results. Obviously, these intermediate results sometimes need to be shuffled – exchanged and/or merge-sorted – across the network to be reduced. In details, the full dataflow is represented in Figure 3.4 using the taxonomy reminded by [37]:

1. *Input reader* – It is in charge of reading origin file of datasets generally by blocks and converts them to key/value pairs.

2. *Map function* – It takes a key/value pair form the input reader, performs the Map on it, outputs the results as key/value pairs.

3. *Combiner function* – (This step is optional.) Actually, it is provided for cases where there is significant repetitions in the intermediate keys produced by each Map task; and for cases where the user's Reduce function is commutative and associative. In such cases, the combiner function performs partial reduction so that pairs with the same key are processed as a group.

4. *Partition function* – As default, a hashing function is used to partition the intermediate keys output from the map tasks to reduce tasks. While this in general provides good balancing, users can still define their own partition function.

5. *Reduce function* – It is called once for each distinct key and applied on the set of its associated values *i.e.* pairs with the same key are processed as a group.

6. *Output Writer* – This step is responsible for writing the output to a stable storage.

The MapReduce paradigm as described provides a new paradigm to deal with distributed dataset. In fact, it proposes to not only consider dataset as distributed and fragmented on each machine but also to develop the computation as small blocks (the Map part) which are finally grouped together (the Reduce part).

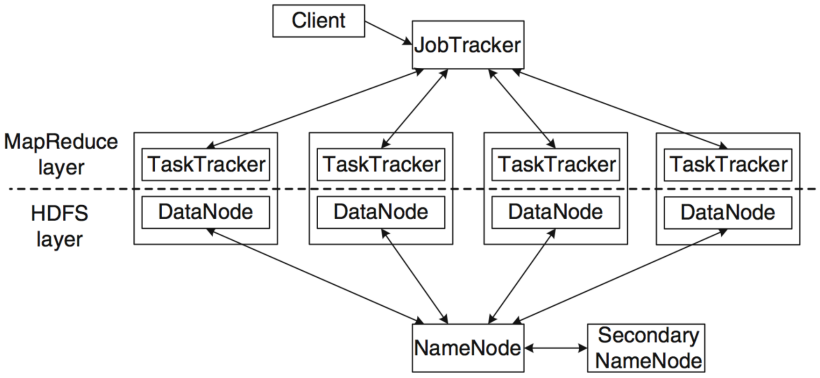


Figure 3.5: Apache Hadoop Architecture [37].

3.2.2 Apache Hadoop

Hadoop is a framework for distributed system based on the Map-Reduce paradigm [34], it is used by numerous evaluators. Hadoop consists of two main parts: the Hadoop distributed file system (HDFS) [81] and a MapReduce library for distributed processing. As illustrated in Figure 3.5, Hadoop consists of a number of different daemons/servers: NameNode, DataNode, and Secondary NameNode for managing HDFS, and JobTracker and TaskTracker for performing MapReduce.

Technically, Files in HDFS are split into a number of large blocks (usually a multiple of 64MB) which are stored on DataNodes. A file is typically distributed over a number of DataNodes in order to facilitate high bandwidth and parallel processing. In order to improve reliability, data blocks in HDFS are replicated and stored on three (default parameter) machines, with one of the replicas in a different rack for increasing availability further. The maintenance of file metadata is handled by a separate NameNode. Such metadata includes mapping from file to block and location (DataNode) of block. The NameNode periodically communicates its metadata to a Secondary NameNode which can be configured to do the task of the NameNode in case of the latters failure.

3.3 Distributed RDF Datastores

Since the RDF is increasingly adopted to model data, managing large volumes of such datasets becomes essential. In addition, the increasing size of these RDF datsets implies more and more often the use of distributed context.

In order to provide efficient SPARQL evaluation for distributed RDF storage, the literature has reviewed several strategies as presented in Figure 3.6. From the angle of their underlying stores, systems can be classified into the following categrories:

- *Federated Systems*: systems warehousing RDF data in a "federation" of centralized RDF stores.
- *Key-Value*: systems using existing "NoSQL" key-value stores
- *Independent*: systems which distributed RDF by themselves
- *Distributed File Systems*: systems relying on a distributed file system such as the HDFS

3.3.1 Federated Strategies

This category comprises systems that exploit in parallel a set of centralized RDF stores distributed among many nodes. These systems have a master/slave architecture, where the master is responsible for partitioning and placing the RDF triples in the slave nodes. Each slave node stores and

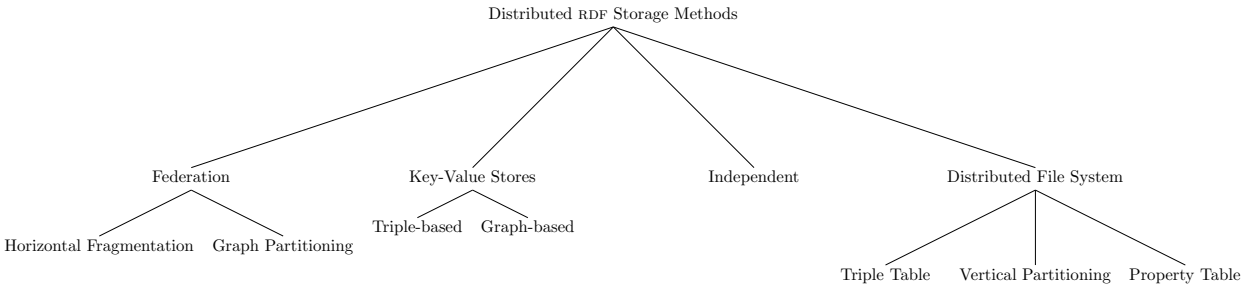


Figure 3.6: Taxonomy of common distributed RDF storage strategies [57].

indexes its local RDF triples in a centralized RDF store. The goal is to partition the RDF data in a way that enables high parallelization during query evaluation while striving to minimize communication among the slave nodes. Such an approach includes [40, 55, 56].

3.3.2 KV-Based Systems

RYA RYA [73] is a native RDF solution leveraging Apache Accumulo that creates three indexes and stores them in Accumulo. Accumulo then sorts and partitions these tables across the nodes, storing data on the HDFS.

3.3.3 DFS-Based Systems

S2RDF S2RDF [78] uses SparkSQL to store RDF triples. SparkSQL [10] is a library built to leverage relational data on top of Apache Spark [87]. It allows users to register files as tables and then to query them using the SQL relational query language. It thus offers a way to set up a distributed relational store, potentially leveraging years of research in relational database systems. S2RDF adopts the vertical partitioning [4] to construct its tables and also computes additional tables based on pre-computation of possible joins representing co-occurrence of a variable in two different fields. Before the evaluation, S2RDF translates SPARQL queries into SQL ones using statistics on orignal data (generated during the preprocessing phase) to order joins by selectivity.

CliqueSquare CliqueSquare [43] is a native RDF solution. The specificity of CliqueSquare lays in trying to reduce the response time by flattening execution plans. Specifically, it implements optimizations whose goal is to minimize the height of the tree of joins in execution plans. It does so in the query optimization phase but also in the way it stores data. Each node is responsible for a set of values storing all triples containing these values as subject, predicate or object (a triple is thus stored, at most, thrice).

PigSPARQL PigSPARQL [77] compiles a SPARQL fragment to PigLatin [70], which is a programming language for distributed systems. PigSPARQL has no actual loading phase. It reads its data directly from the HDFS in the N-Triples W3C standard [3] (*i.e.* a plain text file, with one triple per line with space as the field separator). The PigSPARQL compilation tries to optimize the execution plan through basic writing rules. Such programs are then executed by series of MapReduce jobs.

3.3.4 Independent Solutions

4store 4store[1] is a native RDF solution introduced in [51]. 4store has an index to translate URIs to identifiers, which allows a space-efficient representation of triples. For each predicate it uses

[1] `http://4store.org/`

two indexes (subject to object and object to subject) for optimizing query evaluation. 4store distinguishes two types of cluster nodes: some nodes only store data while others are responsible for parsing, communicating with storage nodes and aggregating the results.

CumulusRDF CumulusRDF[2] [60] relies on Apache Cassandra[3] [61] and mixes two strategies: indexing and hashing. Each triple is hashed and distributed through Cassandra. Additionally the CumulusRDF layer computes indexes to optimize the search of triples satisfying TPs.

CouchBaseRDF CouchBase[4] is not a native RDF solution but a document-oriented NoSQL database system, well-known in the NoSQL world. The specificity of this datastore is that it adopts an in-memory approach where a dataset is distributed on the main memory of the cluster's nodes. This is a limitation because the whole dataset has to fit inside the global RAM – but this speeds up query evaluation. Querying is done by MapReduce rounds on CouchBase controlled by Apache Jena[5], which optimizes the execution plan. CouchBaseRDF [32] transforms CouchBase into an RDF solution. It maps the RDF triples onto JSON documents, each document corresponds to a subject and contains two JSON arrays of the same size: the predicates and the objects. This encoding is used to optimize the retrieval of triples when the subject is fixed. Three views are pre-generated to cover other TPs (when predicate, object or both are fixed values).

3.4 Surveys and Benchmarks

We previously reviewed various technics to store RDF datasets and also to distribute them across clusters of nodes using sometimes popular intermediate tools. For each possible strategy, we also presented at least one representant. It appears that the development of SPARQL evaluators is a very dynamic research field providing each years new methods and solutions. In order to stay up to date, several studies have been yet realized such as reports or surveys to group in lists the various systems of the literature. Moreover, in an attemp to shed light on performances and limitations of current systems, systematic processes such as benchmarks have been created.

3.4.1 Previous Surveys

In the past fifteen years, technical reports and papers have been published to describe, list and compare the various RDF storage solutions. These are milestones that group technics and studies according to different points of view; they obviously represent inventories at a precise moment. Chronologically, we list here some surveys that focus on different aspects of RDF storage and SPARQL evaluation:

- In 2001, Barstow [13] realized one of the first survey of RDF triple stores. He focuses on open-source solutions available at that moment; and look at some of their specificities such as programming language, capacity, performance, APIs, possibility of inferencing... However, he did not provide comparative tests.

- One year later, in [15], Beckett considered the same set of criteria as Barstow and then updated the previous work. Parallely, in [65], Magkanaraki *et al.* reviewed tools whose goals deal with ontologies: processing, accessing and quering them. They described a wide set of tools and query languages thanks to performance figures and comparative analysis.

[2] `http://code.google.com/p/cumulusrdf/`
[3] `http://cassandra.apache.org/`
[4] `http://www.couchbase.com/`
[5] `https://jena.apache.org/`

- In 2003, Beckett *et al.* [16] focused on the use of relational database management systems to store RDF datasets.

- In 2004, Lee also realized a survey of RDF storage approaches in [63].

- Five years later, Stegmaier *et al.* evaluated RDF databases supporting the SPARQL query language in [82]. They reviewed these solutions according to several parameters such as their licenses, their architectures. They also compared their respective interpretation of SPARQL queries using a scalable test dataset.

- More recently, Faye *et al.* listed in 2012 in [39] the various RDF storage approaches mainly used by single-node systems.

- In 2013, Cudré *et al.* realized an empirical study of distributed SPARQL evaluators in [32]. In fact, they took native RDF stores and several NoSQL solutions they adapted to evaluate SPARQL; they then benchmarked them on a common cluster and shared their results.

- Very recently, in 2015, Kaoudi & Manolescu in [57] presented a survey focusing only on RDF in the clouds.

3.4.2 RDF/SPARQL Benchmarks

In the previous sub-section, we established a chronological list of surveys in RDF storage methods and SPARQL evaluators. We notice that only [65, 82, 32] presented also comparative results of evaluator performances, the other studies just review various criteria such as the set of supported features.

Actually, comparative experiments are also a relevant way to rank available SPARQL evaluators. For instance, by loading the same RDF datasets on each system and by querying them after, one can decide which system is the fastest.

Based on that idea, researchers have develop standardized and reproducible RDF/SPARQL benchmarks. These benchmarks are usually made of two parts: first the datasets and second a list of queries which should be evaluated on these datasets; sometimes a testing scenario is also presented, for example, it provides a defined order to execute the queries and suggests that some should be tested several times. Since, the most important concept of them is to offer reproducibility to the community, datasets can often be generated – in a deterministic way – and the list of queries is either pre-defined or generated. It even exists tools to generate RDF "fake" data from an initial dataset that share the same structure *e.g.* GRR [21].

Moreover, these benchmarks are often specialized to test particular fragments of the SPARQL grammar. First of all, they almost always focus on `SPARQL SELECT` queries. In addition, we also notice that the BGP fragment (see Chapter 2 for a more detailed description) constitutes the common base which is always tested in the set of queries and sometimes additional SPARQL keywords such as `OPTIONAL` or `UNION` are part of the patterns.

Practically, the RDF/SPARQL benchmarks usually rank SPARQL evaluators according to the temporal performance of the tested systems. Indeed, they often recommend to pay attention to needed times to load dataset and then to execute each query. Sometimes they also consider the disk footprint of the system. Finally, they may also provide *mixed metrics* where various measurements are aggregated using for instance averages after several computations of the test suite.

In the last fifteen years, a large number of benchmarks have been created and deployed. We provide here a non-exhaustive list of popular benchmarks:

- LUBM [48] is a benchmark proposed in 2005 by the Lehigh University. It focuses on the BGP fragment with 14 SPARQL queries which should be evaluated on generated datasets.

- WatDiv [7] is a more recent benchmark proposed in 2014 by the university of Waterloo. It provides a deterministic RDF data generator. It then also provides sets of SPARQL which should be generated according to 20 query-shapes and the previously generated dataset. These shapes are divided into 4 types: centralized, stared, linear and "snow flake". It strictly focuses on the BGP fragment.

- SP^{2}Bench [79] is settled in the DBLP scenario and comprises both a data generator for creating arbitrarily large DBLP-like documents and a set of carefully designed benchmark queries. It also tests a large fragment of SPARQL with `FILTER`, `OPTIONAL`, `UNION`, the solution modifiers and also three `SPARQL ASK` queries.

- BolowgnaBench [36] provides a framework for evaluating the performance of RDF systems on a real-world context derived from the Bologna process; it strains systems using both analytic and temporal queries; and it models real academic information needs. In terms of SPARQL fragment, it focuses on testing BGPs and also provides queries with select aggregators such as `COUNT` which are part of the standard since the 1.1 version.

- BSBM [20] has been designed to compare performance of native RDF stores with the performance of SPARQL-to-SQL rewriters across architectures. It provides a "query mix" which tests the same SPARQL fragment as SP^{2}Bench excepted the `ASK` but instead it tests also the negation and the `CONSTRUCT`.

- DBPSB [67] – DBPedia SPARQL Benchmark – is a general SPARQL benchmark procedure, which uses the DBpedia [12] knowledge base. The benchmark is based on query-log mining, clustering and SPARQL feature analysis. In contrast to other benchmarks, it performs measurements on actually posed queries against existing RDF data.

- RBench [74] is an application-specific framework to generate RDF benchmarks: it takes an RDF dataset as a template, and generates a set of synthetic datasets with similar characteristics including graph structure and literal labels. RBench then analyzes several features from the given RDF dataset, and uses them to reconstruct a new benchmark graph. A flexible query load generation process is then proposed according to the design of RBench.

It even exists federated projects to develop such benchmarks. For instance, Linked Data Benchmark Council [9] is an european project that aims to develop industry-strength benchmarks for graph and RDF data management systems.

To conclude, we can say that in the last decade, various methods to store RDF datasets have emerged and since distributed systems are more and more usual these storage methods have evolved. In the same time, several surveys and benchmarks have been proposed to rank SPARQL evaluators. In the rest of this study, we will focus on how to efficiently evaluate SPARQL queries in a distributed context.

Part II

Contributions on Efficient Distributed SPARQL Evaluation

Table of Contents

This Part deals with our contributions on efficient distributed SPARQL evaluation. Indeed, we first benchmark state-of-the-art distributed SPARQL evaluators and extract some of their limitations in Chapter 4. Then, we present an evaluator we designed which outperforms state-of-the-art system in Chapter 5. In Chapter 6 we present an extension of the Chapter 4: we extend the set of metrics to fit a distributed context; and thereby offer new perspectives in the development of distributed RDF/SPARQL-based applications based on a multi-criteria approach. In Chapter 7, we consider the particular case of direct SPARQL evaluation introduced in Chapter 6 and presents in detail two SPARQL evaluators we developed: RDFHive and SDE. Finally, in chapter 8, we present a practical example of application where heterogeneous sources of different sizes need to be merged and/or queried sequentially; more particularly, we describe in this Chapter an enriched trip planner.

Chapter 4

Benchmarking Distributed State-of-the-art SPARQL Evaluators on a Common Basis

We introduce in this Chapter a common basis of comparison using various SPARQL evaluators (presented in Chapter 3) that we benchmark on our own cluster.

Contents

With the increasing availability of RDF [53] data, the W3C standard SPARQL language [46] plays a role more important than ever for retrieving and manipulating data. Recent years have witnessed the intensive development of distributed SPARQL evaluators [57] with the purpose of improving the way SPARQL queries are executed on distributed platforms for more efficiency on large RDF datasets.

Two factors heavily contributed to offer a large design space for improving distributed query evaluators. First, the adoption of native data representations for preserving structure (propelled by the so-called "NoSQL" initiatives) offered opportunities for leveraging locality. Second, the seminal results on the MapReduce paradigm [34] triggered a rapid development of infrastructures offering primitives for distributing data and computations [87, 70]. As a result, the current landscape of SPARQL evaluators is very rich, encompassing native RDF systems (*e.g.* 4store [51]), extensions of relational DBMS (*e.g.* S2RDF [78]), extensions of NoSQL systems (*e.g.* CouchBaseRDF [32]). These systems leverage different representations of RDF data for evaluating SPARQL queries, such as *e.g.* vertical partitioning [4] or key-value tables [73]. They also rely on different technologies

	Systems	Underlying Framework	Storage Back-End	Storage Layout	SPARQL Fragment
Independant Datastores	4store	—	Data Fragments	Indexes	SPARQL 1.0
	CumulusRDF	Cassandra	Key-Value store	3 hash and sorted indexes	SPARQL 1.1
	CouchBaseRDF	CouchBase	Buckets	3 views	Basic Graph Pattern
HDFS-based Datastores	RYA	Accumulo	Key-Value store on HDFS	3 sorted indexes	Basic Graph Pattern
	S2RDF	SparkSQL	Tables on HDFS	Extended Vertically Partitioned Files	Basic Graph Pattern
	CliqueSquare	Hadoop	Files on HDFS	Indexes	Basic Graph Pattern
	PigSPARQL	PigLatin	Files on HDFS	N-Triples Files	SPARQL 1.0

Table 4.1: Systems used in our tests.

for distributing subquery computations and for the placement and propagation of RDF triples: some come with their own distribution scheme (*e.g.* 4store [51]), others prefer distributed file systems such as HDFS [81] (*e.g.* RYA [73]), while yet others aim at taking advantage of higher-level frameworks such as PigLatin [70] or Apache Spark [87] (*e.g.* S2RDF [78]). Last but not least, many SPARQL evaluators implement optimizations targeting specific query shapes (*e.g.* CliqueSquare [43] that attempts to flatten execution plans for nested joins). This overall richness and variety in distributed SPARQL evaluation systems make it hard to have a clear global picture of the respective advantages and limitations of each system in practical terms.

In an attempt to shed light on the performances and limitations of current systems, we evaluate a panel of 7 state-of-the-art implementations in the field of distributed SPARQL evaluation. We benchmark them on a common basis, paying attention to reproducibility. We report on a detailed performance description and analysis. We report on lessons learned from our experiments. Our empirical analysis pinpoints the advantages and limitations of current systems for the efficient evaluation of SPARQL queries on a commodity cluster.

The rest of this Chapter is organized as follows. We first briefly list the tested systems in Section 4.1. In Section 4.2, we introduce the methodology and the experimental protocol we used *e.g.* the datasets, the queries. We then review in Section 4.3 the experiences for each store. Finally, we briefly conclude with general observations in Section 4.4.

4.1 Benchmarked datastores

We first list the systems used in our tests we selected, focusing on their particularities for supporting RDF querying. We used several criteria in the selection of the SPARQL evaluators tested. First, we choose to focus on distributed evaluators so that we can consider datasets of more than 1 billion triples which is larger than the typical memory of a single node in a commodity cluster. Furthermore, we retained systems that support at least a minimal fragment of SPARQL composed of conjunctive queries and called the BGP fragment (further detailed in Chapter 2). We focused on open-source systems: we wanted to include some widely used systems to have a well-known basis of comparison, as well as more recent research implementations. We also wanted our candidates to represent the variery and the richness of underlying frameworks, storage layouts, and techniques found, so that we can compare them on a common ground (see discussion in Chapter 3). We finally selected a panel of 7 candidate implementations, presented in Table 4.1.

Table 4.1 also summarizes the characteristics of the systems we used in our tests. We split our panel of 7 implementations into subcategories. The first category, called *independant systems*, gathers systems that distribute data using their own custom methods. In contrast, all the other systems use the well-known HDFS distributed file system [81] for this purpose. HDFS handles the distribution of data across the cluster and its replication. It is a tool included in the Apache Hadoop[1] project which is a framework for distributed systems based on the MapReduce paradigm [34] (more

[1] http://hadoop.apache.org/

details can be found in Chapter 3). We present below the 7 selected systems and briefly remind their main characteristics:

- 4store[2] is a native RDF solution introduced in [51]. 4store has an index to translate URIs to identifiers, which allows a space-efficient representation of triples.
- CumulusRDF[3] [60] relies on Apache Cassandra[4] [61] and mixes two strategies: indexing and hashing. Each triple is hashed and distributed through Cassandra.
- CouchBase[5] is not a native RDF solution but a document-oriented NoSQL database system, well-known in the NoSQL world. The specificity of this datastore is that it adopts an in-memory approach where a dataset is distributed on the main memory of the cluster's nodes.
- RYA [73] is a native RDF solution leveraging Apache Accumulo that creates three indexes and stores them in Accumulo. Accumulo then sorts and partitions these tables across the nodes, storing data on the HDFS.
- S2RDF [78] uses SparkSQL to store RDF triples. SparkSQL [10] is a library built to leverage relational data on top of Apache Spark [87]. It allows users to register files as tables and then to query them using the SQL relational query language.
- CliqueSquare [43] is a native RDF solution. The specificity of CliqueSquare lays in trying to reduce the response time by flattening execution plans. Specifically, it implements optimizations whose goal is to minimize the height of the tree of joins in execution plans.
- PigSPARQL [77] compiles a SPARQL fragment to PigLatin [70], which is a programming language for distributed systems. PigSPARQL has no actual loading phase. It reads its data directly from the HDFS in the N-Triples W3C standard [3].

4.2 Methodology For Experiments

For studying how well the distribution techniques perform, we tested the 10 systems presented in Section 4.1 with queries from two popular benchmarks (LUBM and WatDiv), which we evaluated on several datasets of varying size. We precisely monitored the behavior of each system using several metrics encompassing *e.g.* total time spent, CPU and RAM usage, as well as network traffic. In this Section, we describe our experimental methodology in further details.

4.2.1 Datasets and Queries

As introduced in Section 4.1, we focus here on the Basic Graph Pattern (BGP) fragment which is composed of the set of conjunctive queries. It is also the common fragment supported by all tested stores and thus provides a fair and common basis of comparison.

Also for a fair comparison of the systems introduced in Section 4.1, we decided to rely on third-party benchmarks. The literature about benchmarks is also abundant (see *e.g.* [74] for a recent survey). For the purpose of this study, we selected benchmarks according to two conditions: (1) queries should focus on testing the BGP fragment and (2) the benchmark must be popular enough in order to allow for further comparisons with other related studies and empirical evaluations (such as [32] for instance). In this spirit, we retained the LUBM benchmark[6] [48] and the WatDiv benchmark[7] [7].

[2] http://4store.org/
[3] http://code.google.com/p/cumulusrdf/
[4] http://cassandra.apache.org/
[5] http://www.couchbase.com/
[6] http://swat.cse.lehigh.edu/projects/lubm/
[7] http://dsg.uwaterloo.ca/watdiv/

LUBM is composed of two tools: a determinist parametric RDF triples generator and a set of fourteen queries. Similarly, WatDiv offers a determinist data generator which creates richer datasets than the LUBM one in the sens of the number of classes and predicates, in addition, it also comes with a query generator and a set of twenty query templates. We used several standard LUBM and WatDiv datasets with varying sizes to test the scalability of the compared RDF datastores. Table 4.2 presents the characteristics of datasets we used.

Datasets	**Number of Triples**	**Size**
WatDiv1k	109 million	15 GB
Lubm1k	134 million	23 GB
Lubm10k	1.38 billion	232 GB

Table 4.2: Size of sample datasets.

We evaluated on these datasets the provided LUBM queries and generated the WatDiv queries according to the provided templates. LUBM queries were made to represent real-world queries while remaining in the BGP fragment of SPARQL and with a small data complexity (the size of the answer for a query is always almost linear in the size of the dataset). In addition, in the LUBM query set, we notice that one query is challenging: Q2 since it involves large intermediate results and implies a complex join pattern called "triangular". WatDiv queries compared with LUBM ones involved more predicates and classes. Furthermore, WatDiv developers already group their query templates according to four categories: linear queries (L1-L5), star queries (S1-S7), snowflake-shaped queries (F1-F5) and complex queries (C1-C3).

In addition, we can represent a BGP query by a graph where each node corresponds to a triple pattern and where edges between nodes represent a common variable. As presented respectively in Tables 4.3 & 4.4, LUBM and WatDiv queries can be grouped according to their variable graphs. Moreover, the WatDiv query graphs (Table 4.4) show alternate grouping methods *i.e.* C3, F2 and F4 are all variations around an hexagonal graph.

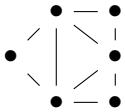
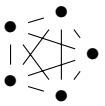
Q6,Q14	Q1,Q3,Q5,Q10,Q11,Q13
Q7,Q12 	Q8 
Q2,Q9 	Q4 

Table 4.3: Variable graphs associated to LUBM queries.

4.2.2 Cluster Setup

Our experiments were conducted on a cluster composed of Virtual Machines (VMs) hosted on two servers. The first server has two processors Intel(R) Xeon(R) CPU E5-2620 cadenced at 2.10 GHz, 96 GigaBytes (GB) of RAM and hosts five VMs. The second server has two processors Intel(R) Xeon(R) CPU E5-2650 cadenced at 2.60GHz with 130 GB of RAM and hosts 6 VMs: 5 dedicated to the computation (like the 5 VM of the first server) plus one special VM that orchestrates the computation. Each VM has dedicated 2 physical cores (thus 4 logical cores), 17 GB of RAM and 6 TeraBytes (TB) of disk. The network allows two VMs to communicate at 125 MegaBytes per Seconds (MB/s) but the total link between the two servers is limited at 110 MB/s. The read and write speeds are 150 MB/s and 40 MB/s shared between the VM on the first server and 115 MB/s and 12 MB/s shared between the VM of the second server.

L3,L4	L1,L2,L5	S6,S7
S2,S3,S4,S5	F1,F3,F5	C1
C3	F2	F4
S1	C2	

Table 4.4: Variable graphs associated to WatDiv queries.

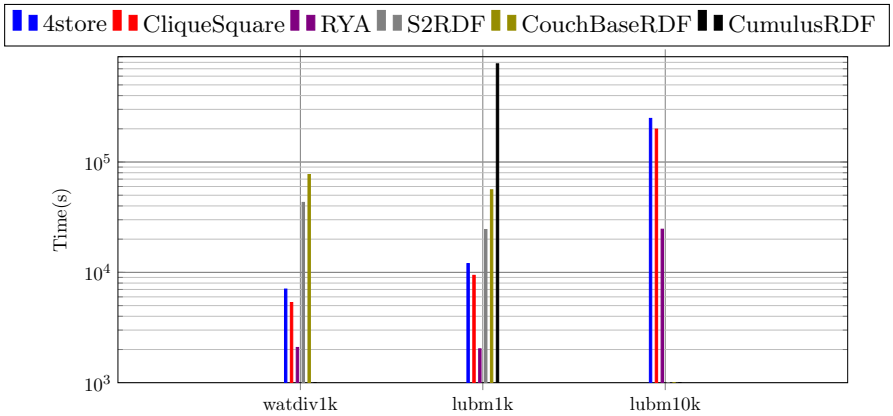


Figure 4.1: Preprocessing Time.

4.2.3 Extensive Experimental Results

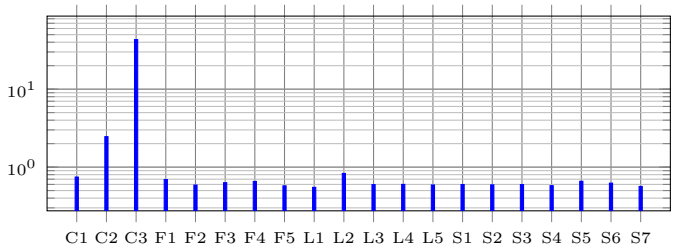
We made our extensive experimental results openly available online with more detailed information. In particular, for reproducibily purposes, we wrote tutorials on how to install and configure the various tested evaluators and report all the versions of the systems we used. We also share measurements and graphs for all the considered metrics and for each node.

`http://tyrex.inria.fr/sparql-comparative/home.html`

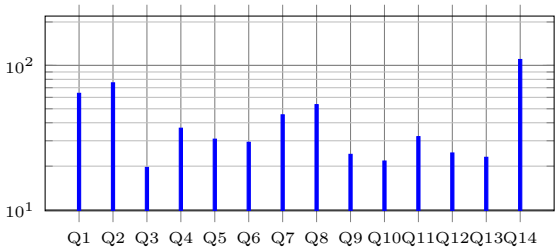
In the rest of the Chapter, we focus on summarizing and discussing the essence of the lessons that we learned from our experiments. In Section 4.3 we report on the overall behavior of each system pushed to the limits during the tests and conclude this Chapter with comparative and more general observations (Section 4.4). In Chapter 6 we will further discuss and develop a comparative analysis guided by practical use cases that imply different requirements.

4.3 Overall Behavior of Systems

In this Section we report on the overall behavior of each tested systems for the three datasets presented in Table 4.2, namely WatDiv1k, Lubm1k and Lubm10k. These datasets constitute appropriate yardsticks for studying how the tested systems behave when the dataset size grows, with



(a) With WatDiv1k (seconds).



(b) With Lubm1k (seconds).

Figure 4.2: 4store Performance.

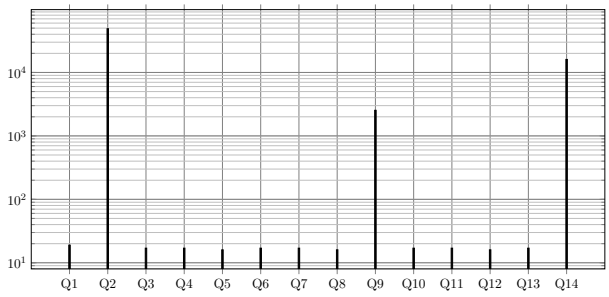


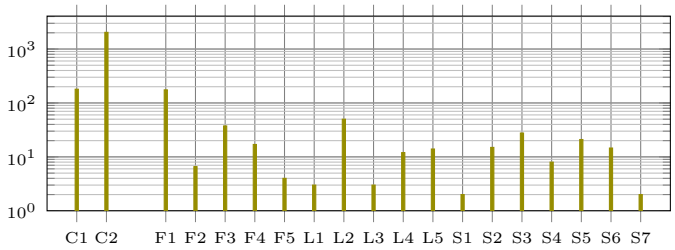
Figure 4.3: CumulusRDF Performance with Lubm1k (seconds).

the characteristics of the cluster used (*cf.* Section 4.2.2). Specifically, the WatDiv1k dataset can still be held in memory of one single VM, while the Lubm1k dataset becomes too large. Lubm10k is even larger than the whole available RAM of the cluster.

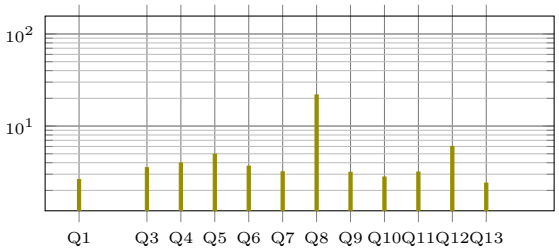
Figure 4.1 aggregates the needed pre-processing times of the 6 selected SPARQL evaluators (since PigSPARQL directly evaluates queries). The rest of the Section is divided according to each evaluators.

4.3.1 4store

4store achieves to load Lubm1k in around 3 hours (Figure 4.1). But it spent nearly three days (69 hours) to ingest the 10 times larger dataset Lubm10k. While the progression was observed to be linear to load smaller datasets (*i.e.* a 2 times larger set was twice longer to load), 4store slowed down with a billion of triples. To execute the whole set of LUBM queries on Lubm1k (Figure 4.2b), 4store never spent more than one minute evaluating each query except Q1, Q2 and Q14 (respectively 64, 75 and 109 seconds). Furthermore, it achieves sub-second response time for WatDiv queries (excepting C2 and C3) with WatDiv1k (Figure 4.2a).



(a) With WatDiv1k (seconds).



(b) With Lubm1k (seconds).

Figure 4.4: CouchBaseRDF Performance.

4.3.2 CumulusRDF

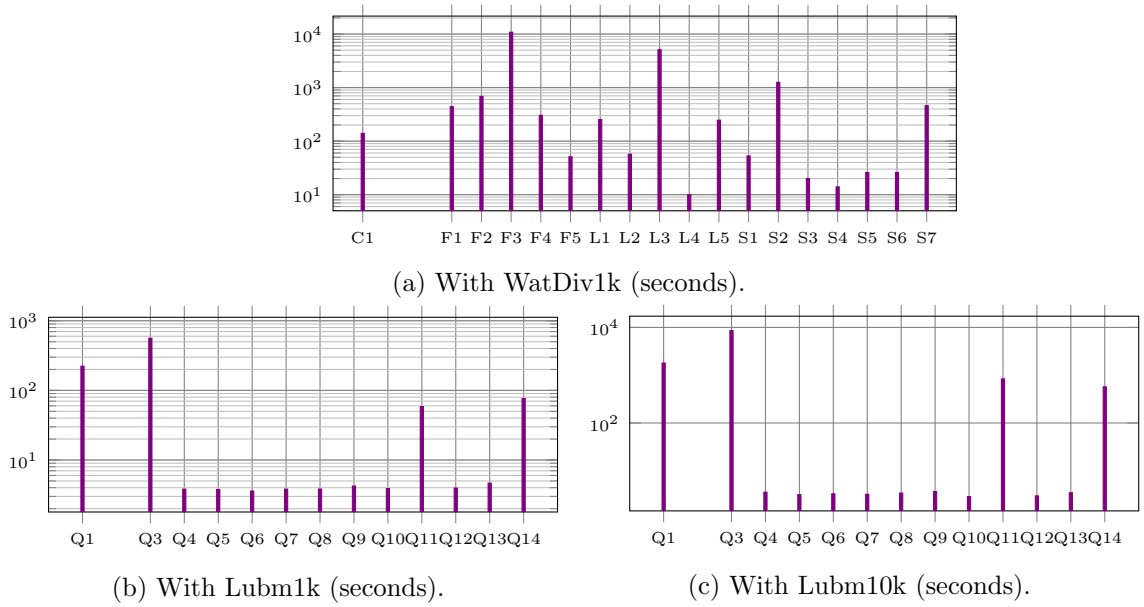
CumulusRDF is very slow to index datasets: it took almost a week only to preprocess Lubm1k (Figure 4.1). By loading smaller datasets (*e.g.* Lubm100 or Lubm10), we notice that the empirical loading time is proportional to the dataset size. That is why we decided not to test it on Lubm10k which is 10 times larger. During the evaluation of the LUBM set of queries on Lubm1k (Figure 4.3), the test of CumulusRDF revealed three points. (1) Q2 and Q9 which are the most difficult queries of the benchmark (see Section 4.2.1) took repectively almost 5000 seconds and 2500 seconds. (2) Q14 answered in 1600 seconds seems to slow CumulusRDF because of its large output. (3) The remaining queries were all evaluated in less than 20 seconds.

4.3.3 CouchBaseRDF

We recall that CouchBaseRDF is an in-memory distributed datastore, which means that datasets are distributed on the main memory of the cluster's nodes. As expected, loading Lubm10k, which is larger than the whole available RAM on the cluster, was impossible. Actually, it crashed our cluster after more than 16 days *i.e.* all the nodes were frozen; and we had to crawl the logs in order to find that it ran out of RAM and SWAP after only indexing nearly one third of the dataset. CouchBaseRDF evaluates quickly queries on Lubm1k (Figure 4.4b), compared to the other evaluators; but it fails answering Q2 and Q14 throwing an exception after two minutes. We also show (Figure 4.4a) that CouchBaseRDF is slow to evaluate C2 (about 2000 seconds) and fails with an exception evaluating C3.

4.3.4 RYA

RYA achieves to load WatDiv1k and Lubm1k in less than one hour and preprocesses Lubm10k in less than 10 (Figure 4.1). However, we note that it needs more preprocessing time with WatDiv1k (15GB) than with Lubm1k (23GB) due to the larger number of predicates WatDiv involves. RYA was not able to answer three queries: C2 & C3 of WatDiv and Q2 of LUBM. In these cases, RYA runs indefinitely without failing or declaring a timeout. To answer the rest of the queries (Figures 4.5a & 4.5b), RYA needs less than 10 seconds for most of the LUBM queries excepting Q1,



(a) With WatDiv1k (seconds).

(b) With Lubm1k (seconds).

(c) With Lubm10k (seconds).

Figure 4.5: RYA Performance.

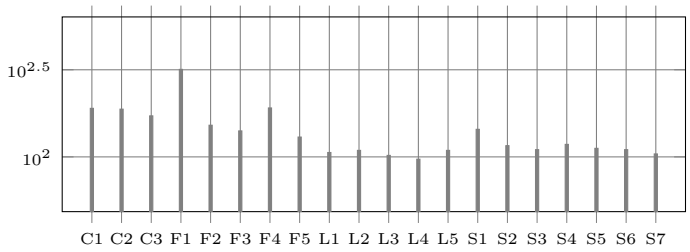
Q3 and Q14. With WatDiv1k, RYA has response times varying over three orders of magnitude *e.g.* L4 which needs 10 seconds and F3 needs 10819. Thanks to its sorted tables (on top of Accumulo), RYA is able to answer quickly queries which involving small intermediate results; therefore, it needs the same amount of time with Lubm10k (Figure 4.5c) than with Lubm1k (Figure 4.5b).

4.3.5 S2RDF

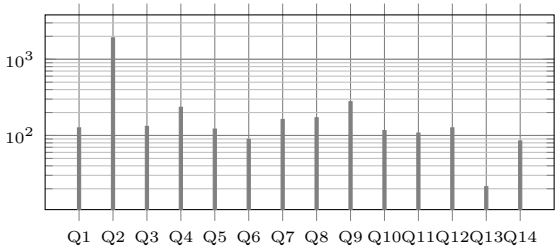
While S2RDF was able to preprocess WatDiv1k and Lubm1k correctly (Figure 4.1), it fails with Lubm10k throwing a memory space exception. Nonetheless, we also notice that preprocessing WatDiv1k was about two times longer than preprocessing Lubm1k; this counterintuitive observation can be explained by the vertical partitioning extension strategy used by S2RDF. Since it computes additional tables based on pre-computation of possible joins, it has to generate more additional table when the number of distinct predicate-object combinations increases. To evaluate WatDiv queries, S2RDF always needs less than 200 seconds excepting F1 (Figure 4.6a) and the average response time is 140 seconds. Figure 4.6b presents the S2RDF results with Lubm1k, we notice that all queries are aswered in less than 300 seconds excepting Q2 which exceeds one thousand seconds due to its large intermediate results that have to be shuffled across the cluster.

4.3.6 CliqueSquare

CliqueSquare achieves to load WatDiv1k, Lubm1k and Lubm10k (Figure 4.1). Figures 4.7b & 4.7c show how its storage model impacts its performances compared to the other evaluators. Actually, having a large number of small files allows CliqueSquare to evaluate the LUBM queries having small intermediate results in the same temporal order of magnitude on Lubm10k as the one needed on Lubm1k (see *e.g.* Q10). We notice that CliqueSquare cannot establish a query plan for the WatDiv queries with its SPARQL parser reporting that the URIs were not "correctly formated". We finally succeeded to evaluate some queries by modifying their syntax as explained in our website. Unfortunately, it appears that we cannot hack queries having at least such a predicate: "<...#type>" (*i.e.* F1, F2, F5, S2, S3, S5, S6 and S7) unless we modify Cliquesquare's source code. Nonetheless, CliqueSquare needs 12 seconds in average to answer each WatDiv linear query, and spends more than one minute to evaluate each complex one (Figure 4.7a).

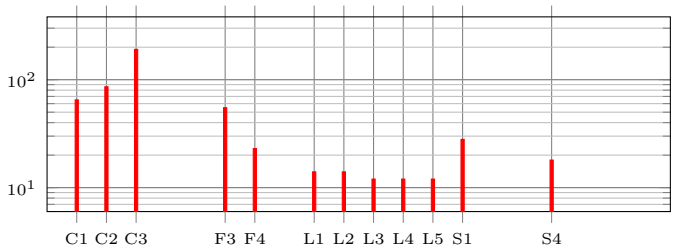


(a) With WatDiv1k (seconds).

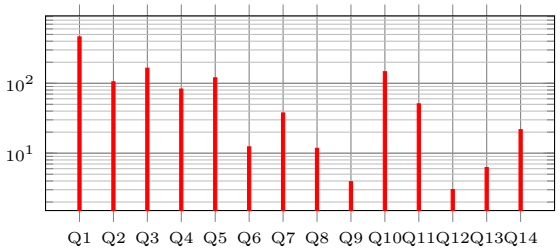


(b) With Lubm1k (seconds).

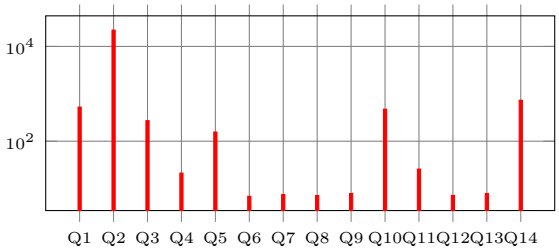
Figure 4.6: S2RDF Performance.



(a) With WatDiv1k (seconds).

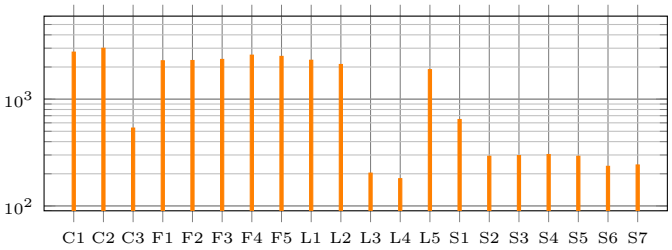


(b) With Lubm1k (seconds).

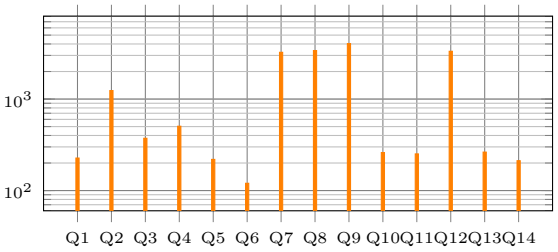


(c) With Lubm10k (seconds).

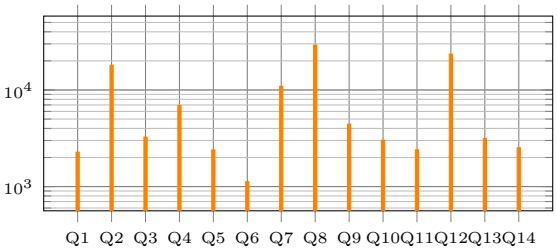
Figure 4.7: CliqueSquare Performance.



(a) With WatDiv1k (seconds).



(b) With Lubm1k (seconds).



(c) With Lubm10k (seconds).

Figure 4.8: PigSPARQL Performance.

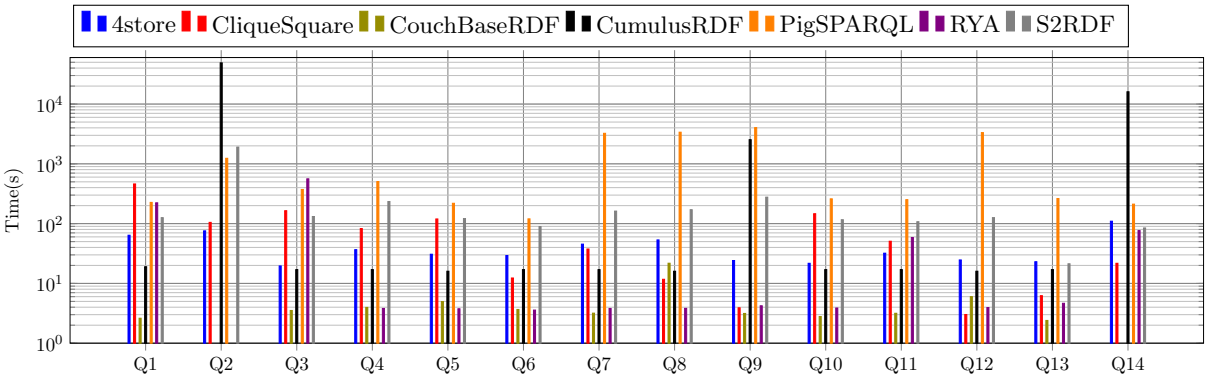


Figure 4.9: Query Response Time with Lubm1k.

4.3.7 PigSPARQL

PigSPARQL evaluates directly the queries after a translation from SPARQL to a PigLatin sequence. Thus, there is no preprocessing phase, we just have to copy the triple file on the HDFS. As shown in Figure 4.8b, PigSPARQL needs more than one thousand seconds to answer queries 2, 7, 8, 9 and 12 on Lubm1k while the other queries take around 200 seconds. We observe the same behaviors when evaluating these queries on Lubm10k (Figure 4.8c). Similarly, the same order of magnitude applies with WatDiv1k (Figure 4.8a).

4.4 General Observations

A first lesson learned is that, for the same query on the same dataset, elapsed times can differ very significantly (the time scale being logarithmic) from one system to another (as shown for instance on Figure 4.9).

Interestingly, we also observe that, even with large datasets, most queries are not harmful *per se*, *i.e.* queries that incurr long running times with some implementations still remain in the "comfort zone" for other implementations, and sometimes even representing a case of demonstration of

efficiency for others. For example, the response times for Q12 with Lubm1k (see Figure 4.9) span more than 3 orders of magnitude. Interestingly and more generally, for each query, there is at least a difference of one order of magnitude between the times spent by the fastest and the slowest evaluators.

4.5 Conclusion

In this Chapter, we updated experimental studies that review distributed SPARQL evaluators. For instance, compared with the work [32] carried out by Cudré *et al.*, we update the list of evaluators since we consider more of them, with more recent ones. It appears that:

1. There are significant performance disparities between evaluators (see *e.g.* Section 4.4).
2. Each evaluator spreads its own results at least over two orders of magnitude. Moreover, the "hard" queries are not the same for one evaluator to an other.
3. We also noticed – practically – that installing such these evaluators can be very complicated and time consuming *e.g.* our tested version of CliqueSquare requires Hadoop version1 while the other evaluators run on version2.

In the next Chapter, we will introduce our own solution (SPARQLGX) to evaluate SPARQL queries in a distributed context. The development is mainly motivated by the idea of simplicity; in fact, we want our system to be both efficient in all cases and easy to install for people who already have installed popular tools such as an HDFS while presenting a simple design.

Chapter 5

SPARQLGX

In this chapter, we introduce SPARQLGX: our implementation of an efficient distributed RDF datastore based on Apache Spark relying on a translation of SPARQL queries into executable Spark code that adopts evaluation strategies according to the storage method used and statistics on data.

Contents

After the performance review of seven various state-of-the-art distributed SPARQL evaluators in Chapter 4, we present in this Chapter our own one named SPARQLGX. The rest is organized as follows: first, the Section 5.1 briefly reminds the needs for such an evaluator. Then, in Section 5.2, we describe SPARQLGX and present additional available tools. Section 5.3 reports on our experimental validation to compare our implementation with other open source HDFS-based RDF systems. Finally, conclude in Section 5.4.

5.1 Motivations

The construction of efficient SPARQL query evaluators faces several challenges. First, RDF datasets are increasingly large, with some already containing more than a billion triples. To handle efficiently this growing amount of data, we need systems to be distributed and to scale. Furthermore, semantic data often have the characteristic of being dynamic (frequently updated). Thus being able to answer quickly after a change in the input data constitutes a very desirable property for a SPARQL evaluator. In this context, we propose SPARQLGX: an engine designed to evaluate SPARQL queries based on Apache Spark [87]: it relies on a compiler of SPARQL conjunctive queries which generates Scala code that is executed by the Spark infrastructure.

The source code of our system is openly available online from the following URL under the CeCILL[1] license:

`https://github.com/tyrex-team/sparqlgx`

[1] CeCILL v2.1: `http://www.cecill.info/index.en.html`

5.2 SPARQLGX: General Architecture

In this Section, we explain how we translate queries from our SPARQL fragment into lower-level Scala code [69] which is directly executable with the Spark API. To this end, after presenting the chosen data storage model, we give a translation into a sequence of Spark-compliant Scala-commands for each operator of the considered fragment.

5.2.1 Data Storage Model

In order to process RDF datasets with Apache Spark, we first have to adopt a convenient storage model on the HDFS. From a "raw" storage (*e.g.* a file in the N-Triple standard which is a simple list of all triples) to complex schemes (*e.g.* involving indexes or B-trees), there are many ways to store RDF data. Any storage choice is a compromise between **(1)** the time required for converting origin data into the target format, **(2)** the total disk-space needed, **(3)** the possible response time improvement induced.

RDF triples have very specific semantics. In a RDF triple (s p o), the predicate p represents the "semantic relationship" between the subject s and the object o. Thus, there are often relatively few distinct predicates compared to the number of distinct subjects or objects. The vertically partitioned architecture introduced by Abadi *et al.* in [4] takes advantage of this observation by storing the triple (s p o) in a file named p whose contents keeps only s and o entries.

(1) Converting RDF data into a vertically partitioned dataset does not involve complex computation: each triple is read once and the pair (subject, object) is appended to the predicate file.

(2) For large datasets with only a few predicates, two URIs are stored instead of three which reduce the memory footprint compared with the input dataset.

(3) Having vertically partitioned data reduces evaluation time of triple patterns whose predicate is a constant (*i.e.* not a variable): searches are limited to the relevant files. In practice, one can observe that most SPARQL queries have triple patterns with a constant predicate. [41] showed that graph patterns where all predicates are constant represent 77.81% of the queries asked to DBpedia and 98.08% of the ones asked to SWDF.

We believe that vertical partitionning is very well suited for RDF: it implies a pass over the data but with only simple computation, reduces the size of the dataset and provides an indexation.

5.2.2 SPARQL Fragment Translation

We compute the solution of a conjunction of TPs recursively. Given a conjunction of n TPs we recursively compute the set of solution for the $n-1$ first TPs and then we combine this set with the solutions of the last TP by joining them on their common variables.

To compute the solutions for a unique TP: when the predicate is a constant, we open the relevant HDFS file using `textFile`; otherwise, we have to open all predicate files. Then, using the constants of the TP, we use a `filter` to keep only the matching elements. Finally, we use the variables names appearing in the TP for variables. For instance, the following TP {?s age 21 .} matching people that are 21 years old is translated into:

```
val tp=sc.textFile("age.txt")
         .filter{case(s,obj)=>obj==21}
```

In order to translate a conjunction of TPs (*i.e.* a BGP), the TPs are joined. Two set of partial solutions are joined using their common variables as a key: `keyBy` in Spark. Joining TPs is then realized with `join` in Spark. For example the following TPs {?s age 21 . ?s gender ?g .} are translated into:

```
val tp1=sc.textFile("age.txt")
          .filter{case(s,obj)=>obj==21}
          .keyBy{case(s,obj)=>s}
val tp2=sc.textFile("gender.txt")
          .keyBy{case(s,g)=>s}
val bgp=tp2.join(tp1).values
```

A join with no common variables corresponds to a cross product (therefore a `cartesian` in Spark). For a conjunction of n TPs we perform $(n-1)$ joins.

The obtained translation (the Scala-code) thus depends on the initial order of TPs since the joins will be perfomed in the same order. This allows us to develop optimizations based on join commutativity such as the ones presented in Section 5.2.4.

5.2.3 SPARQL Fragment Extension

Once the TPs are translated, we use a `map` to retain only the desired fields (*i.e.* the distinguished variables) of the query. At that stage, we can also modify results according to the SPARQL solution modifiers [46] (*e.g.* removing duplicates with `distinct`, sorting with `sortByKey`, returning only few lines with `take`, etc.)

Furthermore, we also easily translate two additional SPARQL keywords: UNION and OPTIONAL, provided they are located at top-level in the WHERE clauses. Indeed, Spark allows to aggregate sets having similar structures with `union` and is also able to add data if possible with `leftOuterJoin`. Thus SPARQLGX natively supports a slight extension (UNIONs and OPTIONALs at top level) of the extensively studied SPARQL fragment made of conjunctions of triple patterns.

5.2.4 Optimized Join Order With Statistics

The evaluation process (using Spark) first evaluates TPs and then joins these subsets according to their common variables; thus, minimizing the intermediate set sizes involved in the join process reduces evaluation time (since communication between workers is then faster). Thereby, statistics on data and information on intermediate results sizes provide useful information that we exploit for optimisation purposes.

Given an RDF dataset $\mathcal{D}$ having T triples, and given a place in an RDF sentence $k \in \{subj, pred, obj\}$, we define the selectivity in $\mathcal{D}$ of an element e located at k as: (1) the occurrence number of e as k in $\mathcal{D}$ if e is a constant; (2) T if e is a variable. We note it $sel^k_{\mathcal{D}}(e)$. Similarly, we define the selectivity of a TP (a b c .) over an RDF dataset $\mathcal{D}$ as: $SEL_{\mathcal{D}}(a,b,c) = \min(sel^{subj}_{\mathcal{D}}(a)\,,\, sel^{pred}_{\mathcal{D}}(b)\,,\, sel^{obj}_{\mathcal{D}}(c))$.

Thereby, to rank each TP, we compute statistics on datasets counting all the distinct subjects, predicates and objects. This is implemented in a compile-time module that sorts TPs in ascending order of their selectivities before they are translated.

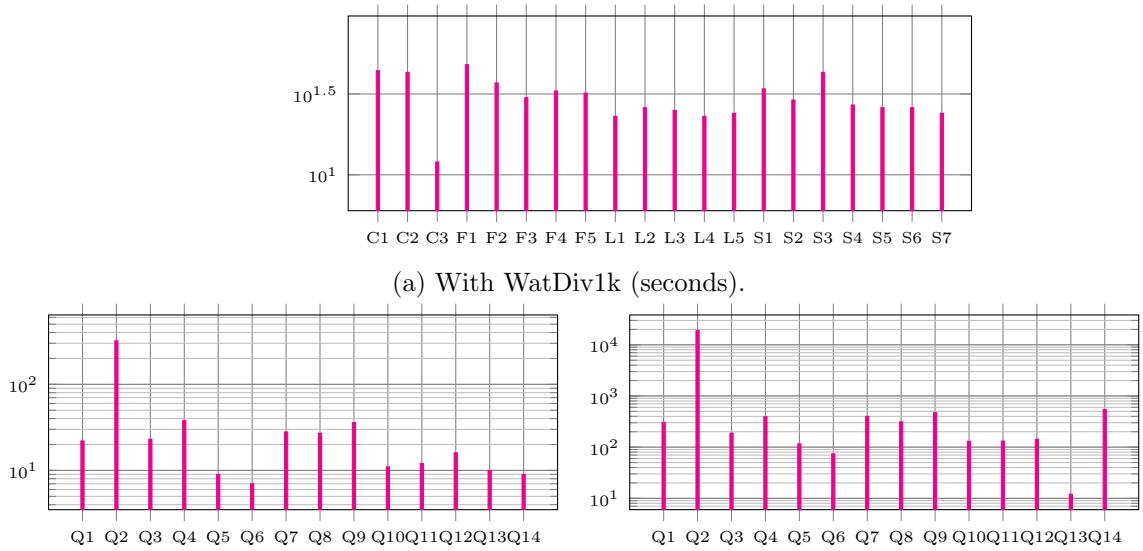
Finally, we also want to avoid cartesian products. Given an ordered list l of TPs we compute a new list l' by repeating the following procedure: remove from l and append to l' the first TP that shares a variable with a TP of l'. If no such TP exists, we take the first.

5.3 Experimental Results

In this Section, we present results obtained after running SPARQLGX on the already introduced cluster in Chapter 4. We first present SPARQLGX performances and then compare them with other HDFS-based evaluators.

Dataset	Number of Triples	Original File Size on HDFS
Watdiv-100M	109 million	46.8 GB
Lubm-1k	134 million	72.0 GB
Lubm-10k	1.38 billion	747 GB

Table 5.1: General Information about Used Datasets.



(a) With WatDiv1k (seconds).

(b) With Lubm1k (seconds).

(c) With Lubm10k (seconds).

Figure 5.1: SPARQLGX Performance.

5.3.1 Performances

Figures 5.1a, 5.1b & 5.1c present the performances with our commodity cluster with the three RDF datasets presented in Table 5.1.

Thanks to its data storage model (*i.e.* Vertical Partitioning), SPARQLGX achieves to pre-process Lubm1k in less than one hour as it does with WatDiv1k (Table 5.2). SPARQLGX pre-processes Lubm10k in about 11 hours.

As shown in Figure 5.1b, all queries but Q2 and Q9 have been evaluated on this dataset in less than 30 seconds. Indeed, these two ones took respectively 250 and 36 seconds. Figure 5.1a shows that SPARQLGX always answer the WatDiv queries in less than one minute, and the average response time is 30 seconds.

5.3.2 Comparison with HDFS-based evaluators

We present here an excerpt of our empirical comparison of our approach with other open source HDFS-based RDF systems. RYA [73] relies on key-value tables using Apache Accumulo[2]. Clique-Square [43] converts queries in a Hadoop list of instructions. S2RDF [78] is a recent tool that allow to load RDF data according to a novel scheme called ExtVP and then to query the relational tables using Apache SparkSQL [10]. Finally, PigSPARQL [77] just translates SPARQL queries into an executable PigLatin [70] instruction sequence.

All experiments are performed on a cluster of 10 Virtual Machines (VM) distributed on two physical machines (each one running 5 of them). The operating system is CentOS-X64 6.6-final. Each VM has 17 GB of memory and 4 cores at 2.1 GHz. We kept the default setting with which

[2]Apache Accumulo: accumulo.apache.org

		Conventional RDF Datastores				Direct Evaluator
		RYA	CliqueSquare	S2RDF	SPARQLGX	PigSPARQL
Watdiv-100M	Preprocessing (minutes)	35	288	718	24	0
	Footprint (GB)	11.0	30.2	15.2	23.6	46.8
	QC (seconds)	TIMEOUT	333	504	**118**	6973
	QF (seconds)	12071	FAIL	771	**182**	9904
	QL (seconds)	5895	**94**	490	119	5670
	QS (seconds)	1892	FAIL	805	**210**	2460
Lubm-1k	Preprocessing (minutes)	34	157	408	55	0
	Footprint (GB)	16.2	55.8	13.1	39.1	72.0
	Q1 (seconds)	192	461	118	**22**	226
	Q2 (seconds)	TIMEOUT	**105**	1599	320	1239
	Q14 (seconds)	66	22	86	**9**	212
Lubm-10k	Preprocessing (minutes)	410	TIMEOUT	FAIL	672	0
	Footprint (GB)	177	403	N/A	407	747
	Q1 (seconds)	1799	524	N/A	**305**	2272
	Q2 (seconds)	TIMEOUT	22093	N/A	**19158**	18029
	Q14 (seconds)	571	731	N/A	**541**	2525

Table 5.2: Compared System Performance.

HDFS is resilient to the loss of two nodes and we do not consider the data import on the HDFS as part of the preprocessing phase.

We compare the presented systems using two popular benchmarks: LUBM [48] and Watdiv [7]. Table 5.1 presents characteristics of the considered datasets. We rely on three metrics to discuss results (Table 5.2): query execution times, preprocessing times (for systems that need to preprocess data), and disk footprints. For space reasons, Table 5.2 presents three Lubm queries: Q1 because it bears large input and high selectivity, Q2 since it has large intermediate results while involving a triangular pattern and Q14 for its simplicity. Moreover, we aggregate Watdiv queries by the categories proposed in the Watdiv paper [7]: 3 complex (QC), 5 snowflake-shaped (QF), 5 linear (QL) and 7 star queries (QS). In Table 5.2 we indicate "TIMEOUT" whenever the process did not complete within a certain amount of time[3]. We indicate "FAIL" whenever the system crashed before this timeout delay. This regroups several kinds of failure such as unability of evaluating queries and also unability of preprocessing the datasets. We indicate "N/A" whenever the task could not be accomplished because of a failure during the preprocessing phase.

Table 5.2 shows that SPARQLGX always answer all tested queries on all tested datasets whereas this is not the case with other conventional RDF datastores which either timeout or fail at some point. In addition, SPARQLGX outperforms several implementations in many cases (also as shown on Table 5.2), while implementing a simple architecture exclusively built on top of open source and publicly available technologies.

5.4 Conclusion

We proposed SPARQLGX: a tool for the efficient evaluation of SPARQL queries on distributed RDF datasets. SPARQL queries are translated into Spark executable code, that attempts to leverage the advantages of the Spark platform in the specific setting of RDF data. We report on practical experiments with our systems that outperform several state-of-the-art Hadoop-reliant systems, while implementing a simple architecture that is easily deployable across a cluster.

[3]We set the timeout delay to 10 hours for the query evaluation stage and to 24 hours for the dataset preprocessing stage.

Chapter 6

A Multi-Criteria Ranking of Distributed SPARQL Evaluators

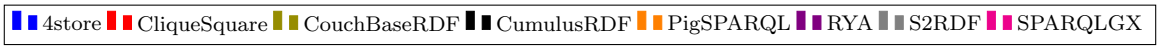
In this Chapter, we provide a new perspective on distributed SPARQL evaluators, based on a 5-criteria ranking (namely velocity, immediacy, dynamicity, parsimony, resiliency). Our suggested set of features provides a more comprehensive description of the behaviors of distributed evaluators when compared to traditional performance metrics. We show how this set of features helps in more accurately evaluating to which extent a given system is appropriate for a given use case. For this purpose, we systematically benchmarked a panel of 8 state-of-the-art implementations. We ranked them using this reading grid to pinpoint the advantages and limitations of current SPARQL evaluation systems.

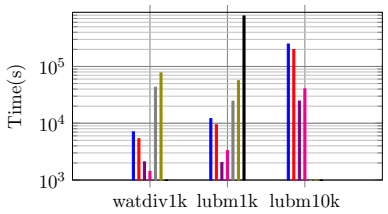
Contents

There exists a variety of advanced SPARQL evaluation systems implementing different architectures for the distribution of data and computations (see *e.g.* Chapter 3). Differences in architectures coupled with specific optimizations for *e.g.* preprocessing and indexing, make these systems incomparable from a purely theoretical perspective. This results in many implementations solving the same problem while exhibiting very different behaviors, not all of them being adapted in any context.

In this Chapter, we extend the scope of metrics considered previously in Chapter 4 and thus we evaluate on the same "commodity" cluster the same panel of 7 state-of-the-art implementations to which we add SPARQLGX already introduced in Chapter 5.

We provide a new perspective on distributed SPARQL evaluators, based on a multi-criteria ranking obtained through extensive experiments. Specifically, we propose a set of five principal features (namely velocity, immediacy, dynamicity, parsimony, resiliency) which we use to rank evaluators. Each system exhibits a particular combination of these features. Similarly, the various requirements of practical use cases can also be decomposed in terms of these features.

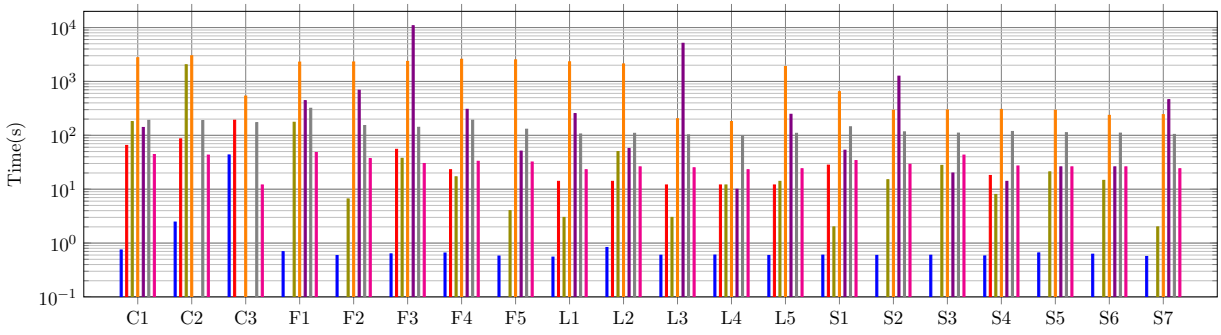




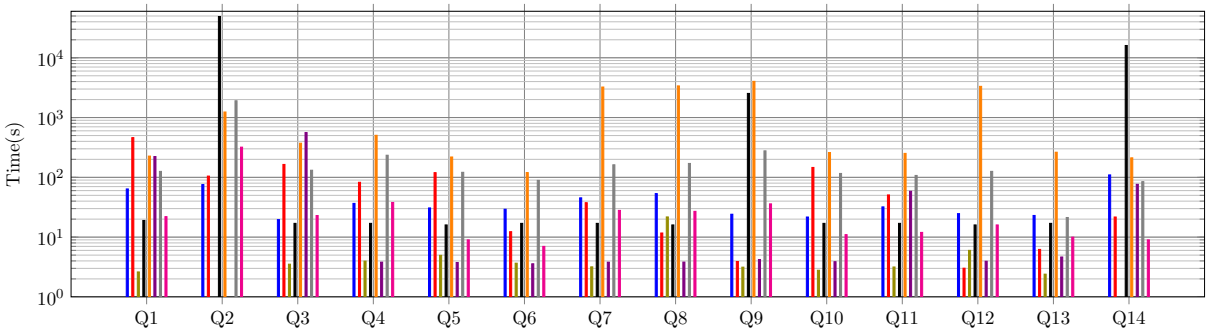
(a) Preprocessing Time.

Evaluator	WatDiv1k	Lubm1k	Lubm10k
CliqueSquare	F1,2,5 & S2,3,5,6,7 Parser	∅	∅
CouchBaseRDF	C3 Failure	Q2,14 Failure	Pre-processing Failure
RYA	C2,3 Timeout	Q2 Timeout	Q2 Timeout
S2RDF	∅	∅	Pre-processing Failure

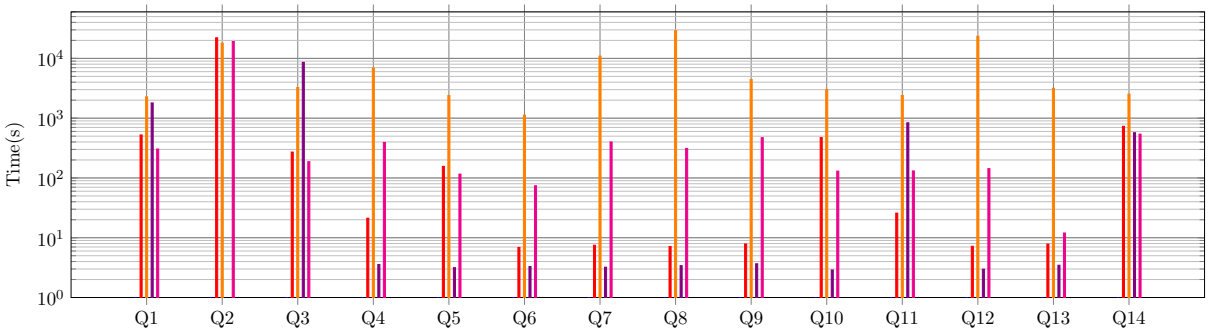
(b) Failure Summary for problematic evaluators.



(c) Query Response Time with WatDiv1k.

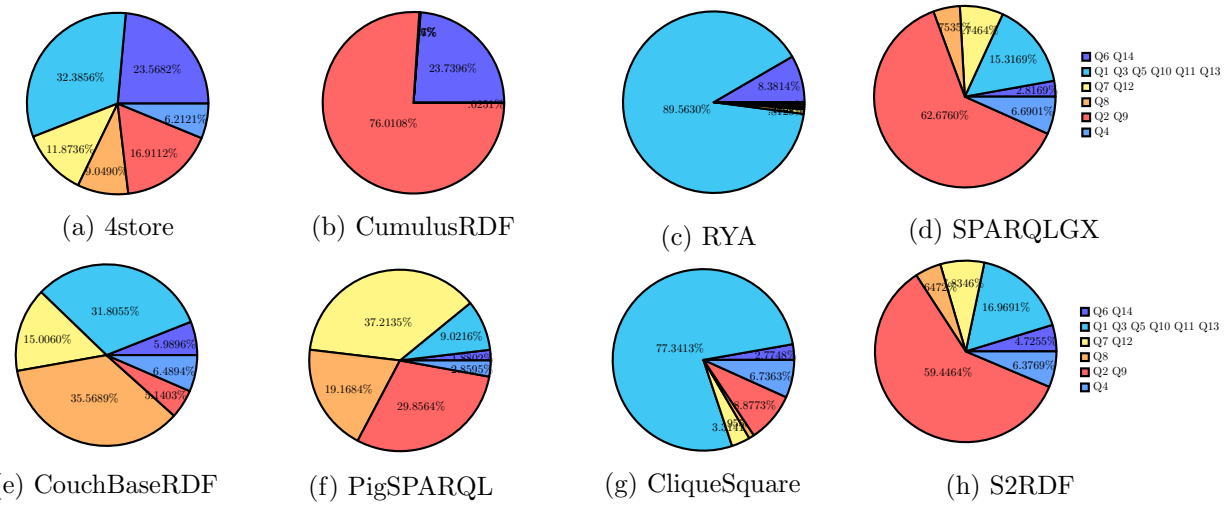


(d) Query Response Time with Lubm1k.



(e) Query Response Time with Lubm10k.

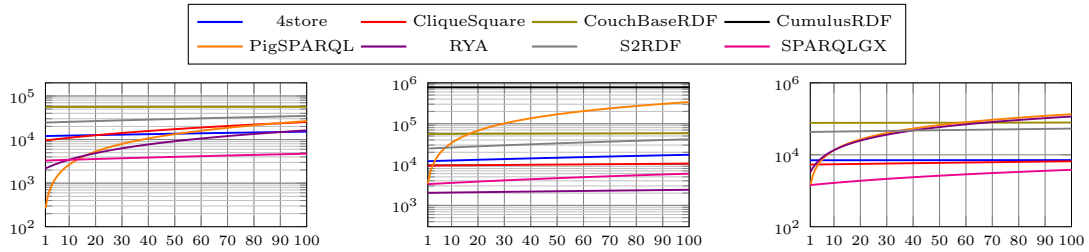
Figure 6.1: Loading and response time with various datasets.



(a) 4store (b) CumulusRDF (c) RYA (d) SPARQLGX

(e) CouchBaseRDF (f) PigSPARQL (g) CliqueSquare (h) S2RDF

Figure 6.2: Time distributions with Lubm1k.



(a) lubm1k: Q1,Q3,Q5,Q10,Q11,Q13 (b) lubm1k: Q8 (c) watdiv1k: L1,L2,L3,L4,L5

Figure 6.3: Tradeoff between preprocessing and query evaluation times (seconds).

Our suggested set of features provides a more comprehensive description of the behavior of a distributed evaluator when compared to traditional performance metrics. We show how it helps in more accurately evaluating to which extent a given system is appropriate for a given use case. For this purpose, we systematically benchmarked a panel of 10 state-of-the-art implementations. We ranked them using this reading grid to pinpoint the advantages and limitations of current SPARQL evaluation systems.

The rest of this Chapter is organized as follows. We first briefly describe the needed metrics we add to monitor the system behaviors. We then present the various use cases we studied. Finally, we discuss the most appropriate systems based on the requirements of different use cases.

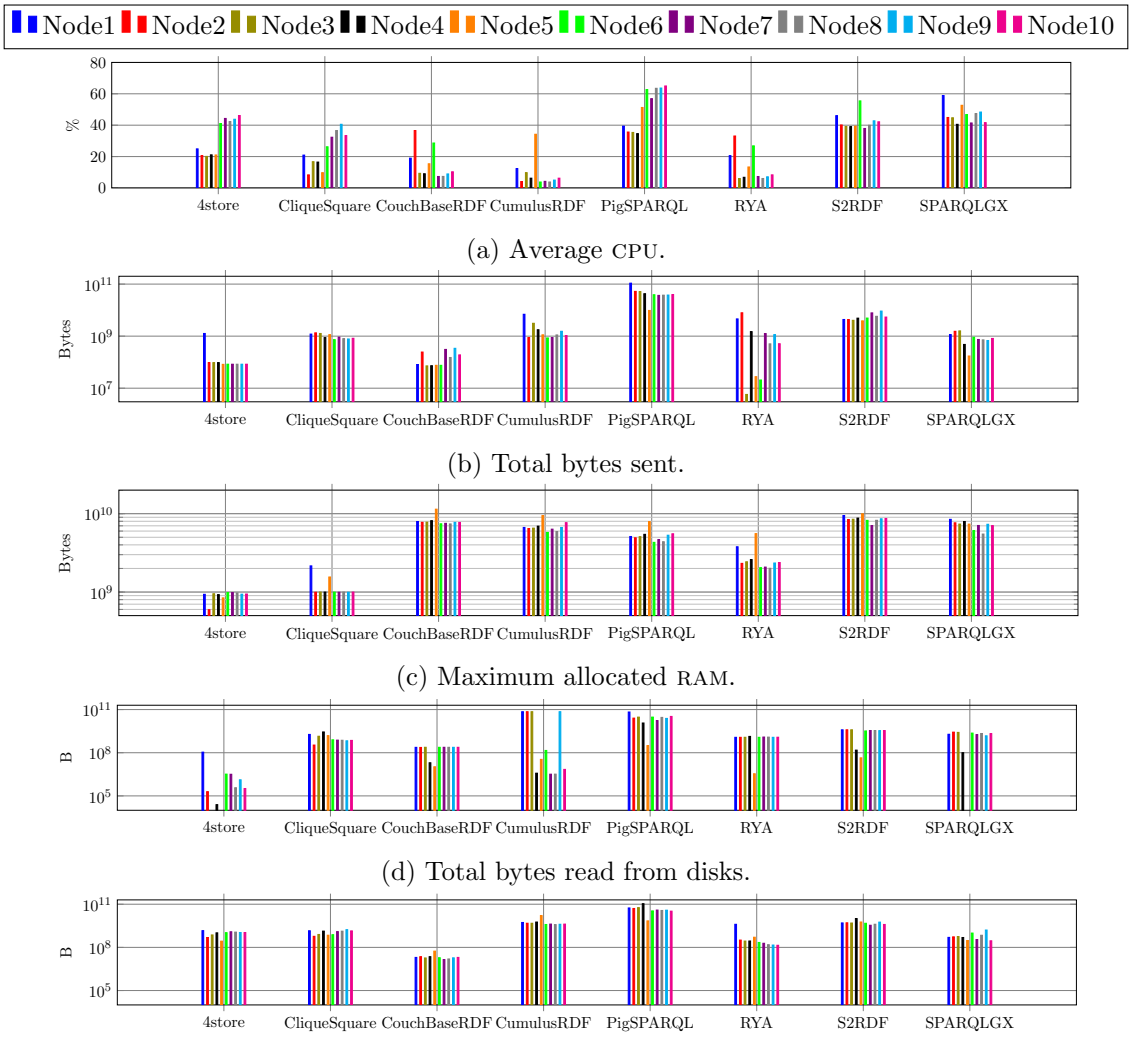
6.1 Extended Set of Metrics

During our tests we monitored each task by measuring not only time spent but a broader set of indicators:

1. *Time (Seconds)*: simply measures the time taken by the system to complete a task.

2. *Disk footprint (Bytes)*: measures the use of disks for a given dataset size including indices and any auxiliary data structures.

3. *Disk activity (Bytes/second)*: measures at each instant the amount of bytes written on and read from the disks during processes.



(a) Average CPU.

(b) Total bytes sent.

(c) Maximum allocated RAM.

(d) Total bytes read from disks.

(e) Total bytes write on disks.

Figure 6.4: CPU, Network and RAM consumptions per node during lubm1k query phase.

4. *Network traffic (Bytes/second)*: measures how much data is exchanged between nodes in the cluster.

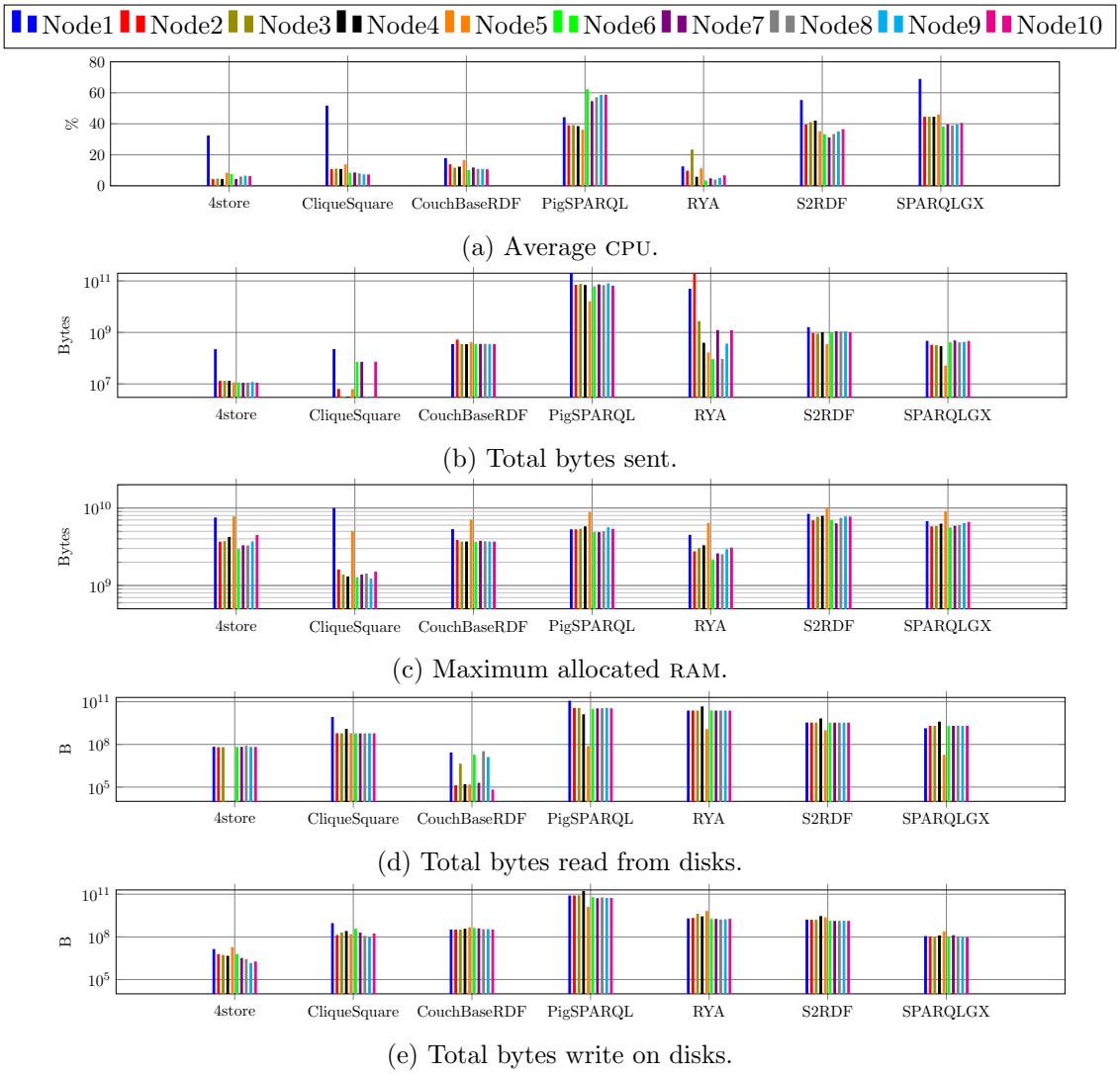
5. *CPU usage (percentage)*: measures how much the CPU is active during the computation.

6. *RAM usage (Bytes)*: measures how much the RAM is used by the computation.

7. *SWAP usage (Bytes)*: measures how much SWAP is used. Such a metric will be particularly measured when the system runs out of RAM and thus be often omitted.

6.2 A Multi-Criteria Reading Grid

The variety of RDF application workloads makes it hard to capture how well a particular system is suited compared to the others in a way based exclusively on time measurements. For instance, consider these five criteria that have different needs and where the main emerging requirement is not the same:



(a) Average CPU.

(b) Total bytes sent.

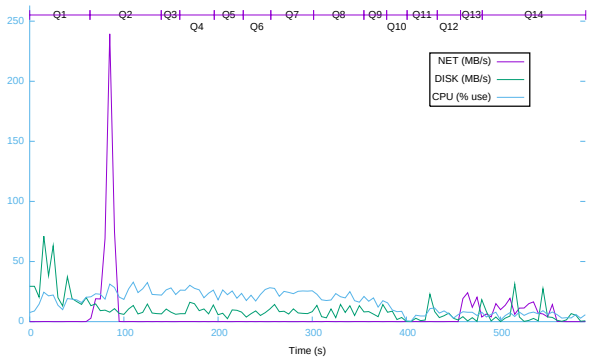
(c) Maximum allocated RAM.

(d) Total bytes read from disks.

(e) Total bytes write on disks.

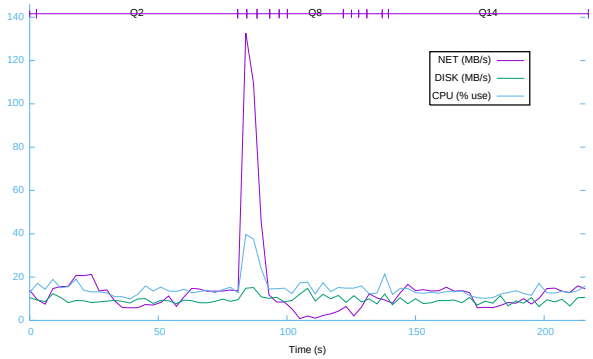
Figure 6.5: CPU, Network and RAM consumptions per node during watdiv1k query phase.

- *Velocity*: applications might favour the fastest possible answers (even if that means storing the whole dataset in RAM, when possible).
- *Immediacy*: applications might need to evaluate some SPARQL queries only once. This is typically the case of some pipeline extraction applications that have to extract data cleaned only once.
- *Dynamicity*: applications might need to deal with dynamic data, requiring to react to frequent data updates. In this case a small preprocessing time (or the capacity to react to updates in an incremental manner) is important.
- *Parsimony*: applications might need to execute queries while minimizing some of the resources, even at the cost of slower answers. This is for example the case of background batch jobs executed on cloud services where the main factors for the pricing of the service are network, CPU and RAM usage.
- *Resiliency*: applications that process very large data sets (spanning accross many machines) with complex queries (taking *e.g.* days to complete) might favour forms of resiliency for trying

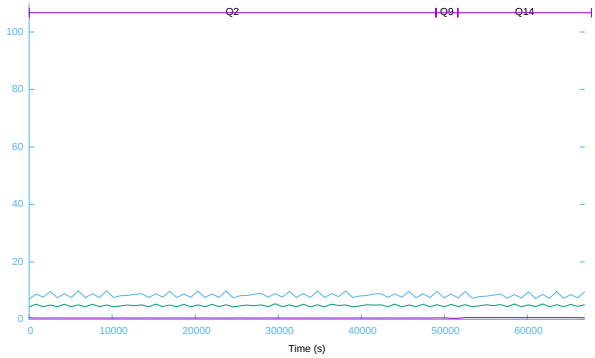


(a) 4store

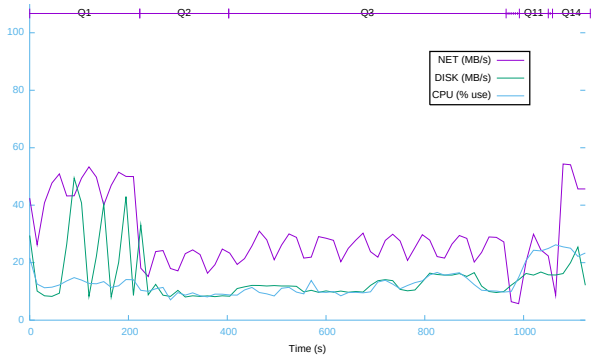
(b) CliqueSquare



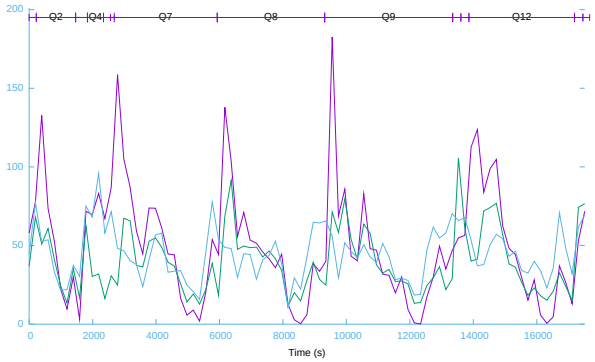
(c) CouchBaseRDF



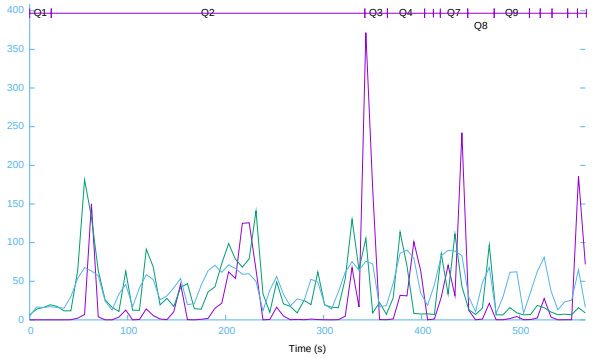
(d) CumulusRDF



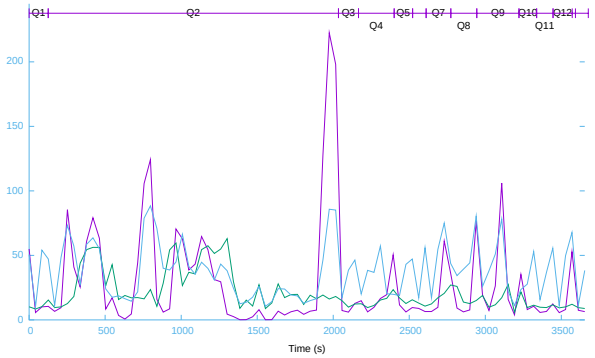
(e) RYA



(f) PigSPARQL



(g) SPARQLGX



(h) S2RDF

Figure 6.6: Resource consumption during Lubm1k query phase.

to avoid as much as possible to recompute everything when a machine fails because it is likely to happen.

Since many applications actually combine these requirements by affecting more or less importance to each, we believe that they represent a good basis on which to compare the tested systems. In this Section, we thus further compare the tested stores by analysing the metrics introduced in Section 6.1 according to the five aforementioned requirements. For the sake of brevity, we will directly refer to these requirements as "velocity", "immediacy", "dynamicity", "parsimony" and "resiliency" in the rest of the paper.

6.3 Velocity *The Faster, The Better*

Figure 6.1d shows the time per query using Lubm1k as dataset for each tested store. The logarithmic scale allows to easily observe the various magnitude orders required to execute queries. It is then possible to notice significant differences between *e.g.* CumulusRDF that needs more than 10^4 seconds to answer Q2 or Q14 while for instance 4store always has response times included in $[10, 100]$ seconds. More generally, it appears that Q2 incurrs the longest response times because of its triangular pattern and its large intermediate results. If we compute the sum of the response times for all the queries of Lubm1k for each evaluator, we notice that our candidates have performances spanning over three orders of magnitude from 568 seconds with SPARQLGX and 67718 seconds with CumulusRDF. Thereby, to execute the whole set of 14 LUBM queries, SPARQLGX and 4store constitute the fastest solutions.

In addition, Figure 6.1d also shows that some stores seem to behave similarly (according to the time metric alone) with some queries *e.g.* PigSPARQL needs the same order of magnitude for evaluating Q2, Q7, Q8, Q9, Q12. That is why we group LUBM queries by their graph variables (introduced in Table 4.3) in Figure 6.2 to represent time distributions for each store, excluding failed queries listed in Figure 6.1b. For instance, Figure 6.2b shows that 99% of the CumulusRDF time is consumed by the evaluation of Q2, Q9 and Q14. This representation also allows to notice similarities between stores, for example we show that because they both rely on Apache Spark, S2RDF (Figure 6.2h) and SPARQLGX (Figure 6.2d) present the same distributions; indeed, their joining method is common even if S2RDF uses the SparkSQL layer. Figure 6.2f shows that PigSPARQL is essentially slow for evaluating Q2,Q7,Q8,Q9,Q12 (about 85% of the time); in fact, we discover that PigSPARQL is slow if there are striclty more than two joins involved in the query.

More generally, this discussion around the variable graphs highlights the RDF storage methods implemented by the considered SPARQL evaluators presented in Table 4.1, classified in literature in *e.g.* [39] and reviewed in Chapter 3. SPARQLGX and S2RDF both share similar pie-charts and vertical partitioning on top of Apache Spark.

6.4 Immediacy *Preprocessing is Investing*

The preprocessing time required before querying can be seen as an investment *i.e.* taking time to preprocess data (load/index) should imply faster query response time, offseting the time spent in preprocessing. To illustrate when the trade-off is really worth, Figure 6.3 presents the preprocessing costs for Lubm1k and WatDiv1k in various cases related to the query types presented in Table 4.3. In other words, we draw on a logarithmic time scale for each evaluator the affine line $y = ax + b$ where a is the average time required to evaluate one of the considered queries and where b is the preprocessing time; for instance in Figure 6.3c, a will represent the average time to evaluate one WatDiv linear queries.

Among the selected competitors, we can distinguish the "direct evaluator" – PigSPARQL – which is capable of evaluating SPARQL queries at no preprocessing cost: it does not require any

preprocessing of RDF data. As shown in Figure 6.3, it appears – as expected – that the systems having complex RDF storage methods have also longer pre-processing phase and therefore are not competitive in this use case; we can list for instance CouchBaseRDF, CliqueSquare, S2RDF or 4store.

These statements are also related to RDF storage approaches; indeed, the more complex it is, the less immediacy-efficient the evaluator is. As a consequence, we can rank for this feature the various storage methods from the best ones: first the schema-carfree triple table of the direct evaluators, next the vertical partitioning, then the key-value table (*e.g.* RYA) and finally the complicated indexing methods.

6.5 Dynamicity *Changing Data*

We now examine how the tested stores can be set up to react to frequent data changes. The W3C proposes an extension of SPARQL to deal with updates[1]. Instead of re-loading all the datasets after each single change, some solutions can be set up to load bulks of updates. To the best of our knowledge, there is no widely-used benchmark dealing exclusively with the SPARQL Update extension. That is why we develop a basic experimental protocol based on both LUBM and WatDiv benchmarks. It can be divided into three steps: **(1)** We load a large dataset *i.e.* Lubm1k (Table 4.2) and evaluate the simple LUBM query Q1 then we measure performances for preprocessing and query evaluation. **(2)** We add a few RDF triples to modify the output of Q1; we run again Q1 and then remove the freshly added triples while measuring the time for each step. **(3)** Finally, we reproduce the previous step with a larger number of triples using WatDiv1 (which contains about one hundred thousand triples) and querying with C1. Although simple, our protocol allows testing the several features such as inserting/deleting a few triples and a large bulk of triples. The benchmarked datastores exhibit various behaviors. First, the direct evaluator (*e.g.* PigSPARQL) evaluate queries without requiring a preprocessing phase. In that case, updating a dataset boils down to editing a file on the HDFS and retriggering query evaluation. Second, other datastores simply do not implement any support (even partial) of updates. This category of stores (*e.g.* S2RDF, CumulusRDF, CouchBaseRDF, RYA or CliqueSquare) thus forces the reprocessing of the whole dataset. Third, some of the benchmarked datastores are able to deal with dynamic datasets *i.e.* 4store and SPARQLGX. 4store implements the SPARQL Update extension whereas SPARQLGX offers a set of primitives to add or delete sets of triples. Moreover, unlike 4store, SPARQLGX is also able to delete in one action a large set of triples, whereas 4store needs to execute several "Delete Data"-processes if the considered set cannot fit in memory.

6.6 Parsimony *Share and Parallelize*

Figure 6.5 shows how each cluster node behaves during the Lubm1k query phase and thus provides an idea of how the evaluators allocate resources across the cluster. Such a visualization also confirms some properties one can guess about evaluators. For example by observing the 4store CPU average usage in Figure 6.5a, we can highlight its storage architecture: the Nodes 6 to 10 are more CPU-active during the process (about 40% of CPU whereas other nodes use about 20%) and thus correspond to the 4store computing nodes while the other ones (excepting the driver on Node1) correspond to the 4store storing nodes. In addition, the number of bytes sent across the network provides clues to identify the evaluator driver nodes (Figure 6.5b) *i.e.* it appears that the Node1 of 4store and RDFHive sends at least 10 times more data than the other nodes (which are receiving). According to several observations made previously (see *e.g.* Section 4.3), we know that the RAM usage can be a bottleneck for SPARQL evaluation. Representing in Figure 6.5c the maximum allocated RAM per node during the Lubm1k query phase, we observe that several evaluators are

[1] SPARLQ updates: `https://www.w3.org/Submission/SPARQL-Update/`

Systems	Lubm1k (GB)	WatDiv1k (GB)
S2RDF	**13.057**	15.150
RYA	16.275	**11.027**
CumulusRDF	20.325	–
4store	20.551	14.390
CouchBaseRDF	37.941	20.559
SPARQLGX	39.057	23.629
CliqueSquare	55.753	90.608
PigSPARQL	72.044	46.797

Table 6.1: Disk Footprints (including replication).

closed to the maximum possible of 16GB per node (see Section 4.2.2): CouchBaseRDF which is an in-memory datastore, CumulusRDF and the two Spark-based evaluators *e.g.* S2RDF and SPARQLGX. On the other hand, 4store and CliqueSquare need in average less than one order of magnitude than it is possible to allocated while being temporally efficient (see *e.g.* Section 6.3).

Figure 6.6 presents resource usages correlated with Lubm1k query evaluation. We give three curves for each evaluator during the Lubm1k query phase: first, the network traffic (sent and received bytes); second, the disk activity (read and write bytes); third, the CPU usage. Moreover, we also divide the time dimension according the needed response times of LUBM queries to observe the resource consumption during one designated query at a glance. We observe that Network and Disk peaks are often synchronous, which means the evaluator reads and transmits or receives and saves data. These correlations are especially observed with the direct evaluators since they have to read at least once the whole dataset to evaluate a SPARQL query and also have to shuffle intermediate results to join them (see *e.g.* Figures 6.6f). In addition, we also remark that thanks to their storage models, 4store CliqueSquare or CouchBaseRDF never have to read large amounts of data and we can only observe network peaks when the query has large intermediate results or outputs such as Q14 for example (see *e.g.* Figures 6.6a, 6.6c & 6.6b).

Paying attention to resource consumption thereby provides information on the real evaluator behaviors. Actually, we found that some systems that dominate in previous use cases (*e.g* SPARQLGX for Velocity or PigSPARQL for Immediacy) are in fact costly for the cluster in terms of RAM allocation of CPU average usage. Moreover, we also highlight that the Spark-based evaluators have a selfish behavior by using as much resources as possible in order to provide an answer as quickly as possible. As a conclusion, if one needs to run concurrent processes while evaluating SPARQL queries (*e.g.* running a SQL service or data processing pipelines at the same time), one should rather prefer evaluators whose data storage models are optimized such as 4store or CliqueSquare.

6.7 Resiliency *Having Duplicates*

Data Resiliency When an application processes a very large dataset stored across many machines, it is interesting for the system to implement some level of tolerance in case a datanode is lost. To implement data resilience, stores typically replicate data across the cluster which implies a larger disk footprint. For our experiments, we stick to the default replication parameters. As a consequence, the HDFS-based systems have their data replicated twice and provide some level of data resilience. Table 6.1 presents the effective disk footprints (including replication) with Lubm1k and WatDiv1k where the HDFS-based systems are outlined in gray. Due to their preprocessing methods, we note that S2RDF and CliqueSquare need more disk space to store WatDiv1k than Lubm1k whereas this last one is larger (see Table 4.2). Furthermore, counterintuitively, it appears that evaluators having replicated data can have lighter disk footprints than not-replicated ones *e.g* S2RDF and RYA versus CouchBaseRDF.

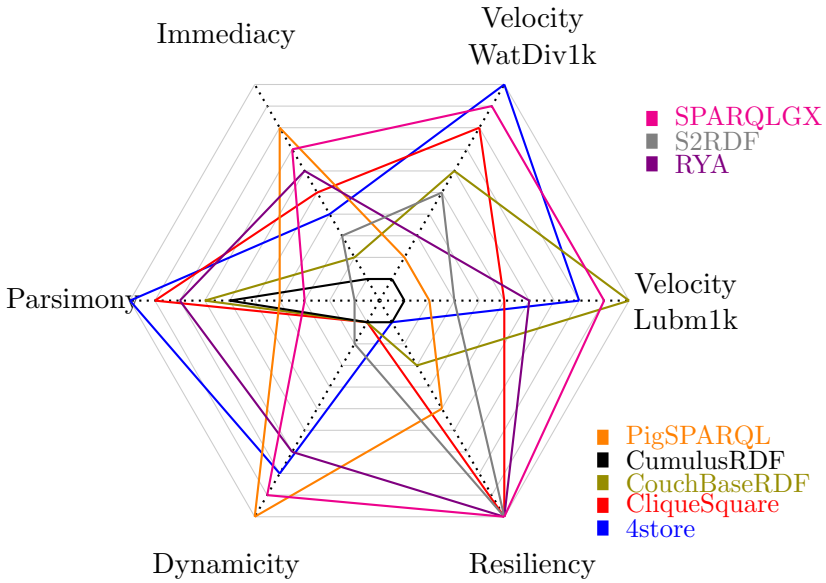


Figure 6.7: System Ranking (farthest is better).

Computation Resiliency If an application has to evaluate complex queries (taking *e.g.* days), it is interesting for the system not to be forced to compute everything from scratch whenever a machine becomes unreachable. This situation is liketly to happen for a variety of reasons (*e.g.* reboot, failure, network latency). The tested systems exhibit several behaviours when a machine fails during computation. For stores having no data replication, the loss of any machine can stop the computation if the lost data fragment is mandatory; thus some stores fail when a machine is lost: 4store and CumulusRDF; whereas CouchBaseRDF adopts another method waiting seven minutes until the return of the machine. More generally, the HDFS-based triplestores cannot lose mandatory fragments of data, thereby RDFHive, SPARQLGX, RYA, and CliqueSquare still succeed when one (or even two) machine fails during computation; however, PigSPARQL waits indefinitely the return of the lost partition. For stores having a master/slave structure *e.g.* SPARQLGX, the loss of the node hosting the master process prevents any result to be obtained. From our tests, only two different methods successfully faced a loss of worker nodes: (1) waiting for their returns *e.g.* CouchBaseRDF and PigSPARQL; (2) using the remaining nodes and benefiting from data replication *e.g.* CliqueSquare, RYA, S2RDF, SPARQLGX.

6.8 Cost & Ranking

As real applications actually combine the 5 previously presented use cases while often adding monetary considerations, we introduce Table 6.2 and Figure 6.7 to offer more hindsight.

Table 6.2 presents an estimation of experiment costs using the MS-Azure cloud platform[2] *i.e.* we simulate prices considering 10 "A4" instances which are close to our VMs in terms of RAM; each one costs $0.480 an hour (U.S. dollars). Estimations are then computed using the following formula (which excludes Network traffic and data generation/import costs): $10 \times price \times benchTime$, where $benchTime$ is the sum of the preprocessing time and the required time by a given number of rounds of successfully answered query ($load + query \times round$). Table 6.2 thus presents costs for 1 and 100 rounds which match with studied features respectively the immediacy and the velocity. Our model shows that costs can be spread over several orders of magnitude. Particularly, it estimates the SPARQLGX and RYA costs evaluating LUBM queries at less than $3 executing one round while

[2]Costs are computed using prices listed as of October 2016. `https://azure.microsoft.com/en-us/pricing/`

	1 round		100 rounds	
Systems	**WatDiv1k**	**Lubm1k**	**WatDiv1k**	**Lubm1k**
4store	$9.48	$16.81	$17.01	$94.74
CliqueSquare	$7.81	$14.17	$77.77	$176.12
CouchBaseRDF	$106.29	$74.80	$453.77	$82.88
CumulusRDF	–	$1125.15	–	$10063.93
PigSPARQL	$36.44	$23.55	$3644.66	$2355.46
RYA	$29.40	$3.98	$2665.31	$130.05
S2RDF	$61.17	$37.53	$433.28	$522.95
SPARQLGX	$2.69	$5.14	$83.08	$80.12

Table 6.2: Cost Estimations (U.S. dollars).

considering 100 rounds 4store and SPARQLGX are the most profitable. More generally, this cost estimation gives additional clues to elect an evaluator including real (monetary) considerations in the selection process.

Figure 6.7 presents a Kiviat chart in which the tested systems are ranked, based on Lubm1k and WatDiv1k according to all the use cases already discussed in this Chapter. More particularly, evaluator ranks on the two "velocity" axes (one for Lubm1k and one for WatDiv1k) are based on average response time considering only successful queries. This representation gives at a glance clues to select an evaluator. For instance it appears that 4store is especially relevant when velocity and parsimony are important and less importance is given to resiliency. SPARQLGX also appears as a reasonnable choice when all criteria (including its potential cost on a cloud platform) but parsimony matter.

6.9 Conclusion

We conducted an empirical evaluation of 8 state-of-the-art distributed SPARQL evaluators on a common basis[3]. By considering a full set of metrics, we improve on traditional empirical studies which usually focus exclusively on temporal considerations. We proposed five new dimensions of comparison that help in clarifying the limitations and advantages of SPARQL evaluators according to use cases with different requirements. In the next Chapter, we will present SPARQL evaluators we made which are especially designed for some dimensions.

[3]We present in this Chapter an excerpt of our experimental results; details are openly available from `http://tyrex.inria.fr/sparql-comparative/`

Chapter 7

SPARQL Direct Evaluation

After, the comparative analysis (see Chapter 6) and after the introduction of SPARQLGX (see Chapter 5), we present in this Chapter methods which mainly tackle the "immediacy" criteria. The first one named RDFHive is based on a relational management system and the second one named SDE is a SPARQLGX variation. We validate our approaches thanks to empirical experiments which were realized using the same cluster and the same evaluators as in Chapters 4, 5 & 6.

Contents

As introduced in Chapter 6, in some cases, users might only need to evaluate a query once *e.g.* to extract a subset of the origin dataset. In such conditions, a SPARQL evaluator is efficient: if it is "quickly ready" to evaluate, and if it has "fast" answer response times. In this use case, called *immediacy*, the preprocessing time before evaluating SPARQL queries can be seen as an investment.

Actually, to compare evaluators, we analyze the cost of preprocessing time regarding the execution time of SPARQL queries. In other words, condidering an evaluator i, we draw the following affine:

$$T_i(n) = Q_i.n + L_i$$

where Q_i represents in seconds the needed time to evaluate once the set of considered SPARQL queries with the evaluator i, n stands for the number of repetitions, and L_i the time to preprocess the dataset. Thereby, to order evaluators, we compare these affines and *e.g.* search for their intersections.

As a consequence, in this Chapter, since "the faster is a preprocessing, the quicker is an evaluator ready", we particularly look at solution having no preprocessing case. In other words, we analyse SPARQL evaluators which directly origin datasets. The rest of this Chapter is divided as follows. We start by introducing such evaluators we made: RDFHive in Section 7.1 and SDE in Section 7.2. Then, we improve the comparative analysis done in Chapter 6 adding as new elements the performances of these two evaluators in Section 7.3. Next in Section 7.4, we also consider the "parsimony" criteria introduced previously in Chapter 6. Finally, we conclude in Section 7.5.

7.1 RDFHive: a Relational Solution

As explained more in details in Chapter 3, several SPARQL evaluators are not RDF native since they rely on relational database management systems RDBMS to store their datasets. Such solutions then

SelectQuery	:=	(prefix)*
		`SELECT` (`REDUCED`|`DISTINCT`)? ("`*`"|(var)+)
		`WHERE` { (tp) (" . "tp)* }
		(`LIMIT` (digit)*)? (`OFFSET` (digit)*)?
prefix	:=	`PREFIX` (alphanum)*: <(alphanum)*>
var	:=	("?"|"$")(alphanum)*
tp	:=	(var|(alphanum)*) (var|(alphanum)*) (var|(alphanum)*)
digit	:=	[0-9]
alphanum	:=	[a-z|A-Z|0-9]

Figure 7.1: RDFHive Supported SPARQL Fragment.

evaluate SPARQL queries after translating them into SQL [**?**]. Even if SPARQL and SQL semantics are not the same [**?**], it is still possible to translate the basic graph pattern (BGP) fragment of SPARQL into correct SQL. That is why, in order to benefit from years of research and development in the relational database field, building reational-based SPARQL evaluators might be relevant.

In 2009, since traditional relational data warehouses could not scale, Thusoo *et al.* proposed Apache Hive [83]. In a nutshell, Hive is an open-source data warehousing solution built on-top of Apache Hadoop [**?**]. As a consequence, it takes as file system the HDFS and converts SQL (technically Hive-QL – but the fragment we consider allow us to use the exact SQL syntax –) queries in sequences of MapReduce jobs executed directly on Hadoop. Therefore, Apache Hive allows to query large datasets distributed across cluster of nodes using a relational language while providing resiliency thanks to Hadoop.

RDFHive. In this context, we propose and share RDFHive: a simple implementation of a distributed RDF datastore benefiting from Apache Hive. RDFHive is designed to leverage existing Hadoop infrastructures for evaluating SPARQL queries. RDFHive relies on a translation of SPARQL queries into SQL queries that Hive is able to evaluate. The sources of RDFHive are openly available under the CeCILL[1] license from:

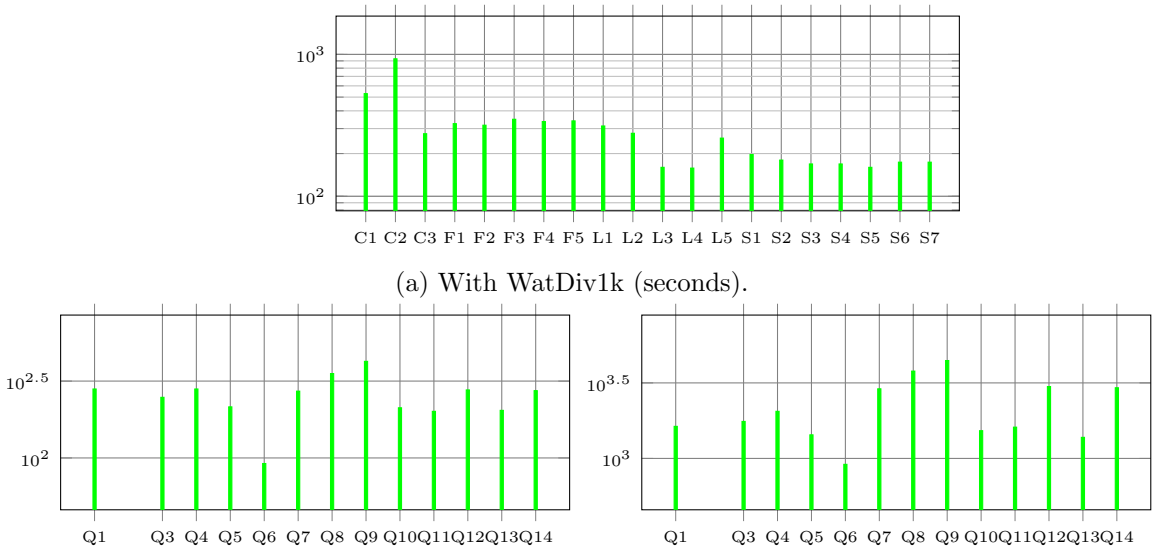
`https://github.com/tyrex-team/rdfhive`

Storage Method. Since, Apache Hive is a distributed RDBMS, its standard structure is a table. To load an RDF dataset into the Hive's warehouse, we need to translate it into a relational table. Looking at the various RDF syntaxes we introduced in Chapter 1, we found that the N-Triples format [3] already have a table-like shape: the first column for the subject, the second one for the predicate and the third one for the object of each RDF statement.

Even if several methods of RDF partitioning can be considered – *e.g.* the vertical one [4] used by SPARQLGX (see Chapter 5) – RDFHive sees an N-Triples RDF file as an already structured relational table having three fields: one for each column. This strategy allows then RDFHive to instantly load such a file (which should prior be stored on the HDFS) as a database of one single table.

Supported SPARQL Fragment. Figure 7.1 presents the SPARQL syntax supported by RDFHive. It basically allows `SELECT` queries (see for more details Chapter 2) restricted to the BGP fragment *i.e.* the clauses in the `WHERE` block should be triple patterns. Additionally, some solution modifiers are also taken into account: `REDUCED`, `DISTINCT`, `LIMIT` and `OFFSET`. Finally, for clarity purposes, SPARQL syntax shortcuts such as `PREFIX` are also allowed.

[1] CeCILL v2.1: `http://www.cecill.info/index.en.html`



(a) With WatDiv1k (seconds).

(b) With Lubm1k (seconds).

(c) With Lubm10k (seconds).

Figure 7.2: RDFHive performance

Evaluation Process. RDFHive directly evaluates SPARQL queries – since there is no preprocessing step, indeed an RDF triple file is seen by Hive as a three-column table –; thus, it simply have to translate SPARQL queries according to this scheme. After a fast re-writting the query according to the potentially defined prefixes, the BGP translation is straightforward and can be divided into three parts. Firstly, RDFHive translates the `WHERE` clauses to form both the `FROM` and `WHERE` statements of a classic SQL query. Secondly, RDFHive translates the beginning of the query *i.e.* the SQL `SELECT` using the distinguished definition of the SPARQL query. Thirdly, it finishes considering the potential solution modifiers.

While the latter two steps of the translation process are simple, the first one is the one where optimizations can be done. Actually, RDFHive translates a list of triple patterns starting sequentially and thereby, changing the order of triple patterns changes the translation output while obviously having the same result. For example, the following queries are two representations of the same result: in this case, their triple patterns have been switched

```
SELECT ?X WHERE { ?X a b . ?X c ?Y }

SELECT t1.subj FROM test.triples AS t1
  JOIN test.triples AS t2 ON t2.subj=t1.subj
WHERE t1.pred='a'
  AND t1.obj='b'
  AND t2.pred='c'
```

```
SELECT ?X WHERE { ?X c ?Y . ?X a b }

SELECT t1.subj FROM test.triples AS t1
  JOIN test.triples AS t2 ON t2.subj=t1.subj
WHERE t1.pred='c'
  AND t2.pred='a'
  AND t2.obj='b'
```

The outputed translations change in the SQL `WHERE` clauses without changing the expected result. However, it can occur that initial triple pattern lists are not optimized at all for RDFHive, and they force it to evaluate costly cross products. This case happens if the translator encounters only new variables in a triple pattern, in such a situation since it cannot join it with a previous one, it realizes a cross product. For instance:

```
SELECT ?X WHERE { ?X a b . ?Y c d . ?X e ?Y }
```

```
SELECT t1.subj
FROM test.triples AS t1 , test.triples AS t2
  JOIN test.triples AS t3 ON t3.subj=t1.subj
WHERE t1.pred='a' AND t1.obj='b'
  AND t2.pred='c' AND t2.obj='d'
  AND t3.pred='e' AND t3.obj=t2.subj
```

```
SELECT ?X WHERE { ?X e ?Y . ?X a b . ?Y c d }
```

```
SELECT t1.subj
FROM test.triples AS t1
  JOIN test.triples AS t2 ON t2.subj=t1.subj
  JOIN test.triples AS t3 ON t3.subj=t1.obj
WHERE t1.pred='e'
  AND t2.pred='a' AND t2.obj='b'
  AND t3.pred='c' AND t3.obj='d'
```

In the first case, the first two triple patterns seem independant to RDFHive since the third one is still ignored; thus, it cross products them. In the second case, two joins are done to group the subresults of each triple pattern. Moreover, because of the selected storage method, cross products on the single table are costly and slow down the evaluation process (see below). In order to avoid time-greedy cross products, RDFHive re-writes (this default behavior that can be disabled off) if possible the SPARQL queries prior to the translation process so that the triple pattern list offers already encountered variables from the second clause.

Technical Details. RDFHive is all implemented in *bash* in only 300 lines. Practically, it presents several *bash* scripts: one to declare a new N-Triples RDF file as a database, one to remove that binding and one to evaluate SPARQL queries according to the supported fragment (see Figure 7.1). The *bash* implementation allows to use RDFHive "out-of-the-box"; it only needs Apache Hive (on top of an HDFS) to be installed on the cluster. It is therefore a quickly ready direct SPARQL evaluator.

Performances. Since RDFHive only needs a triple file loaded on the HDFS to start evaluating queries, we do not consider the database definition from the origin file.

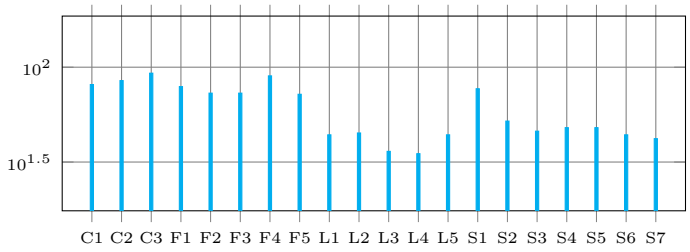
It appears that RDFHive was unable to answer Q2 of LUBM *i.e.* no matter the time allowed, it could not finish the evaluation. On Lubm1k (Figure 7.2b), we also notice that each remaining query is evaluated on Lubm1k in a 200 to 450 seconds period with a 256-second average response time. Similarly (Figure 7.2a), RDFHive has 289-second average response time with WatDiv1k.

7.2 SDE: an Apache Spark Alternative

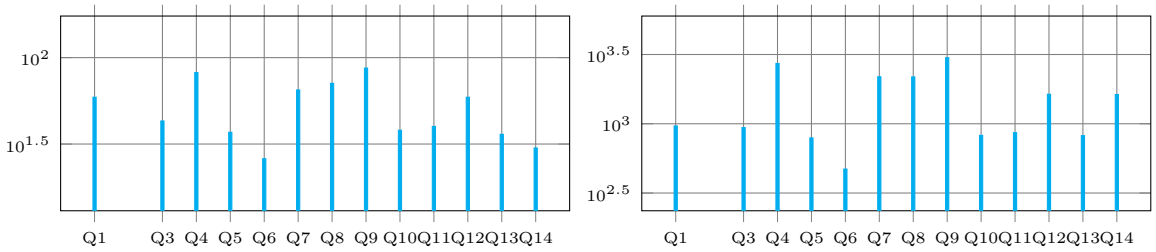
A variant from SPARQLGX. Our tool to evaluate SPARQL queries over distributed RDF datasets *i.e.* SPARQLGX (see Chapter 5 for more details) is base on-top of Apache Spark after preprocessing RDF data. However, as presented in Chapter 6 in certain situations, data might be dynamic (*e.g.* subject to updates) and/or users might only need to evaluate a single query *i.e.* immediacy and dynamicity use cases. In such cases, it is interesting to limit as much as possible both the preprocessing time and the query evaluation time. That is why we built a SPARQLGX extension able to directly evaluate SPARQL queries.

To do so *i.e.* to take the original triple file as source, we only have to modify in our translation process the way we treat TPs to change our storage model. Instead of searching in predicate files, we directly use the initial file; and the rest of the translation process remains the same. We call this variant of our evaluator the "direct evaluator" or SDE. Technically, instead of applying a `textFile` on a chosen two-column predicate file as in SPARQLGX, SDE always applies `textFile` on the three-column N-Triples origin file; the rest of the translation process remains the same. The sources are also openly available from the same SPARQLGX repository:

`https://github.com/tyrex-team/sparqlgx`

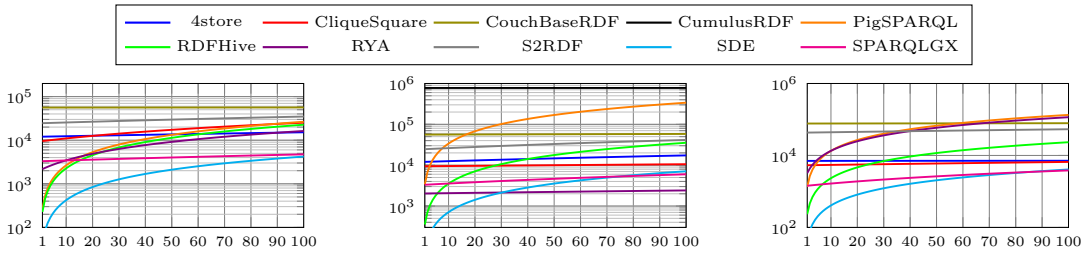


(a) With WatDiv1k (seconds).



(b) With Lubm1k (seconds). (c) With Lubm10k (seconds).

Figure 7.3: SDE performance



(a) lubm1k: Q1,Q3,Q5,Q10,Q11,Q13 (b) lubm1k: Q8 (c) watdiv1k: L1,L2,L3,L4,L5

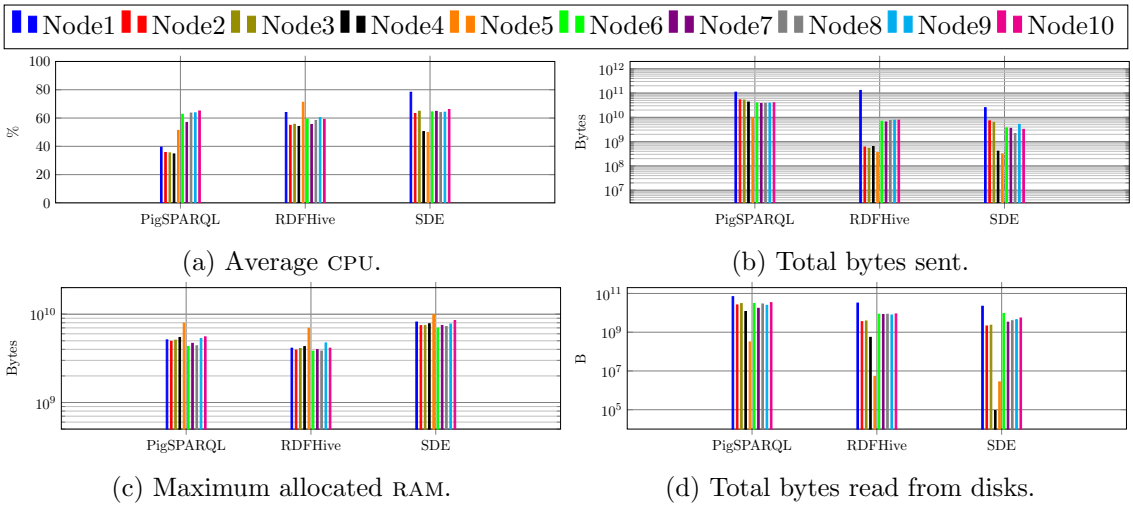
Figure 7.4: Tradeoff between preprocessing and query evaluation times (seconds).

Translation Optimizations. Similarly to SPARQLGX and RDFHive, and since SDE processes the triple patterns linearly during the translation step, modifying the order modifies the output Scala code. As a consequence, an additional step of re-writing can be done to avoid is possible cross-products *i.e.* `cartesian` functions in Apache Spark library. However unlike SPARQLGX, since there is no pre-processing step, SDE never considers statistics on RDF data to optimize its join plan.

Performances. Since SDE is a SPARQL direct evaluator, it does not need any preprocessing time to ingest datasets. Its average response times with WatDiv1k, Lubm1k and Lubm10k (Figures 7.3a, 7.3b & 7.3c) are respectively 60, 51 and 1460 seconds. We observe that the average response time with Lubm10k is about 28 times larger than the one with Lubm1k (which is 10 times larger) indeed Q4, Q7, Q8, Q9, Q12 and Q14 do not perform well because of their large intermediate results.

7.3 Direct Evaluators Versus Conventional Ones

To illustrate when the trade-off is really worth, Figure 7.4 presents the preprocessing costs for Lubm1k and WatDiv1k in various cases related to the query types presented in Table 4.3. In other words, we draw on a logarithmic time scale for each evaluator the affine line $y = ax + b$ where a is the average time required to evaluate one of the considered queries and where b is the preprocessing



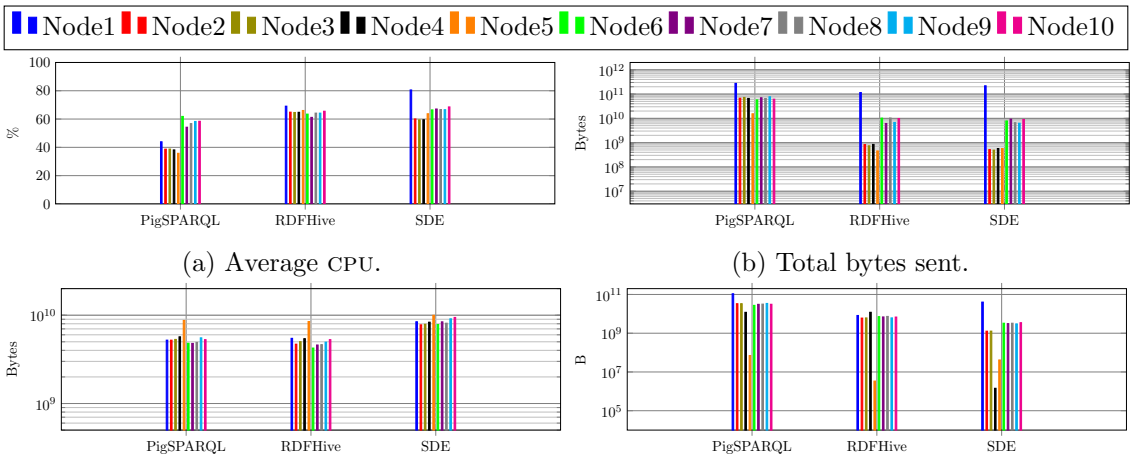
(a) Average CPU.

(b) Total bytes sent.

(c) Maximum allocated RAM.

(d) Total bytes read from disks.

Figure 7.5: CPU, Network and RAM consumptions per node during lubm1k query phase.



(a) Average CPU.

(b) Total bytes sent.

(c) Maximum allocated RAM.

(d) Total bytes read from disks.

Figure 7.6: CPU, Network and RAM consumptions per node during watdiv1k query phase.

time; for instance in Figure 7.4c, a will represent the average time to evaluate one WatDiv linear query. We take back the results already present in Chapter 6 and add the new direct evaluators we built.

Among competitors, we distinguish the set of "direct evaluators" (see Table 4.1) that are capable of evaluating SPARQL queries at no preprocessing cost (they do not require any preprocessing of RDF data): PigSPARQL, RDFHive and SDE. As shown in Figure 7.4, SDE outperforms all the other datastores if less than 20 queries are evaluated. Beyond this threshold, SPARQLGX or RYA become more interesting. In addition, we also notice that in some cases (for instance Q8, see Figure 7.4b) PigSPARQL provide worse performances than RYA or SPARQLGX all the time.

As a consequence, we can consider that our goals are reached since we wanted to design evaluators able to be ready faster than conventional stores and able to answer one single query quickly.

7.4 Resource Consumption and Direct Evaluation

We present in the previous Sections 7.1 & 7.2 the two SPARQL direct evaluators we developed: RDFHive and SDE; we also show how each one performs with popular benchmarks. In addition, we compare them with an other direct evaluator *e.g.* PigSPARQL and with "conventional" ones already introduced in Chapter 4. In this Section, we extend the discussion we had in Chapter 6 dealing particularly with resource consumption of the SPARQL direct evaluators.

Figures 7.5 & 7.6 present the behaviors of the three direct evaluators we observed. These Figures show how each node has been used during LUBM and WatDive query phases and highlight four metrics: the CPU consumption, the amounts of bytes sent and read on disks and the maximum RAM allocation.

We note (Figures 7.5b,7.5d, 7.6b & 7.6d) that PigSPARQL sends more bytes across the network and also reads more data on disks than SDE and RDFHive, while it has the lighest CPU consumption. It means that PigLatin – which is underneath PigSPARQL – caches less intermediate results since the source files are exactly the same raw N-Triples files.

This previous idea is also validated by the observations made with RAM; actually, SDE is the biggest consumer of memory during the query evaluation (see *e.g.* Figure 7.5c). Interestingly, RDFHive is the most parsimonious RAM consumer of the benchmarked direct evaluators thanks to Apache Hive.

As a section-conclusion, we remarked that the temporal efficiencies of RDFHive and SDE are not free: they need large amounts of available resources of the cluster such as the network or the memory.

7.5 Conclusion

The two systems we developed – RDFHive & SDE – both achieve to evaluate SPARQL queries without pre-processing RDF datasets. It appears, as expected, that they are faster than conventional evaluators (which need to treate data prior to evaluate) if they have to evaluate once less than a dozen of queries. As a conclusion, we show that the Triple Table RDF storage model presented in Chapter 3 combined with light distributed tools (such as Apache Hive or Spark) allow to efficiently perform according to the "immediacy" criteria.

Chapter 8

Smart Trip Alternatives

In the previous Chapters, we introduced, benchmarked and analysed several distributed SPARQL evaluators. In addition, we extend the scope of comparison to new dimensions and even designed very specific evaluators to fit with particular use cases.
Now, thanks to an example application, we enlarge our focusing to encompass new types of data sources. More particularly, we present, in this Chapter, a system that automatically computes smart trip alternatives between two cities in the world. To do so, it searches points of interest in large semantic datasets taking into account the set of accessible areas around each possible layover, from which it automatically selects a few relevant options. It can then elect two feasible alternatives while displaying their differences with respect to the default trip. This Chapter gives us the possibility to integrate some of the tools we made into a case of wider use.

Contents

In this Chapter, we present a "real-world" application which depends on multiple heterogeneous data sources. Indeed, as a practical example of the themes introduced in the previous Chapters of this study, we develop an air-trip planner which enriches stopovers with touristic places. The rest of this Chapter is organized as follows. First in we present the general context of such an application in Section 8.1. Then, we detail the GTFS store we made to handle public transport schedules in Section 8.2. In Section 8.3, we present how we designed our system around a scalable infrastructure for supporting the mass of worldwide GTFS information, how we leverage various data sources in heterogeneous formats (*e.g.* RDF, JSON, XML, GTFS, *etc.*) for semantic enrichment of information, and how we encode constraints and heuristics for the efficient selection of smart trip options. In Section 8.4 we illustrate the use of our novel system in a real-world setting before concluding in Section 8.5.

8.1 Motivations & Context

In trip planning, it is very common to query for flight combinations according to criteria such as shortest total duration, or cheapest combination for instance. Resulting routes often include inescapable waiting times at airports between connecting flights. Instead, other trip alternatives

might reveal much more interesting, such as those setting an appropriate time between connections to allow for a specific activity that leverages the local environment. For instance, when travelling from Lyon to Singapore, the shortest duration criterion yields a stopover in Dubai with a waiting time of 3 hours. This might represent significant waiting time, while being too short for an activity outside of the airport. Instead, it might be more interesting to slightly defer the connection (by *e.g.* 2 hours) and obtain enough time to enjoy Dubai's city on the way to Singapore; or to save a hotel night at destination when the initial connection arrives in the late evening at the profit of daytime spent at an alternate stopover such as Frankfurt. Choosing among such alternatives requires additional planning efforts to make sure that *e.g.* points of interest can effectively and conveniently be reached in the allowed time frame, attractions of interest are open, etc. This additional effort is particularly significant when the user is not aware of local possibilities at all viable stopovers.

We introduce a system that computes and suggests smart trip alternatives automatically, given any two origin and destination cities in the world. Our system explores large universes of semantically-checked possibilities in the set of all viable layovers, from which it automatically selects a few relevant options. Our system finally suggests two feasible smart trip alternatives while displaying their differences with respect to the default trip with *e.g.* shortest duration. Thus our system does not require any additional user input when compared to a system such as Google Maps [1] or Rome2Rio [2].

Our system leverages the increasing availability of open city transportation data (in *e.g.* GTFS [45]), and combines them with flight information as well with external data sources for the selection of *e.g.* particularly remarkable points of interests.

8.2 GTFS Store

A Native Store. General Transit Feed Specification (*i.e.* GTFS) [45] is an open-source format that can be used by transit agencies to share and publish their service schedules. This standard has been first published by Google in 2006. Recently, it has been adopted by the transit industry as a standard for sharing schedule data. For example, there exists sites where datasets can be uploaded. One of the most famous is the GTFS Data Exchange[1] wich provides data from at least 950 transit agencies. More specifically, one GTFS dataset consists of several text files presented as CSV. Data can be of two types: required fields or optional ones. Required data includes information regarding transit (*agency, stops, routes, trips, stop times...*) whereas *fare*, *transfers* and also *route alignment* are not mandatory.

In order to take advantage of the increasing amount of data conforming to this specification, we set up a store able to process GTFS. Given a dataset, we first read the various files of the standard in order to load relevant pieces of information. Thereafter, queries can be sent to the store. We next provide further details about the background system and its optimisations.

The implementation of our GTFS store is openly available from:

`https://github.com/tyrex-team/gtfs-store`

Technical Optimizations. First of all, since the amount of GTFS data is expected to increase, we need a system capable of scaling in terms of the number of transport agencies offering their data. Therefore, we implement our system on top of a cluster of machines and we use Apache Spark [87] as a library to interface the Hadoop distributed file system (HDFS) [22] with our algorithms. Additionally, the resiliency in case of machines falling comes for free. Actually, our GTFS store is resilient thanks to the distributed replication policy of data *i.e.* the replication factor is set to three.

[1] http://www.gtfs-data-exchange.com/

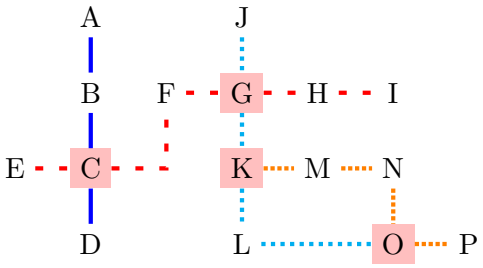


Figure 8.1: Example of 4 routes with 4 connection stops.

As mentioned previously, a GTFS dataset is composed of several files each one providing specific information about transit feeds. Then, to describe the methods and optimization techniques used, we describe in more details what types of information are conveyed by GTFS. Actually, each kind of data is registered into a special file. To build our solution, we extract relevant fields in several files. First, stop names and GPS coordinates are written with their identifiers in *stops.txt*. Then, in GTFS, journeys are decomposed into a hierarchical structure. Indeed, the transit networks is divided into **routes**, **trips** and finally into **paths**. A *route* corresponds to the intuitive idea of a line *i.e.* it always has the same stops. Second, a *trip* is a sequence of two or more stops occuring at a specific time; thus a route is a group of trips *e.g.* a trip could be one of the daily train of a certain route. Finally, a *path* is the smallest division of a trip, in other words it represents a fragment of a trip corresponding to the transit between two consecutive stations. Each level of information is written in a dedicated file of the specification: *routes.txt*, *trips.txt* and *stop_times.txt*.

During the loading step, we adopt a specific policy taking advantage of Apache Spark and of our distributed file system (HDFS). Actually, we register in a file all **direct pairs** with departure and arrival times. With such an arrangement, *filters* can be processed quickly. For instance the following list of paths extracted from *stop_times.txt*:

```
(trip_id,arr,dep,stop_id,stop_sequence,...)
trip0,08:00:00,08:00:00,stop0,0,...
trip0,09:00:00,09:05:00,stop1,1,...
trip0,10:00:00,10:05:00,stop2,2,...
trip0,11:00:00,11:05:00,stop3,3,...
```

would become the following pairs after loading:

```
stop0,stop1,08:00:00,09:00:00
stop0,stop2,08:00:00,10:00:00
stop0,stop3,08:00:00,11:00:00
stop1,stop2,09:05:00,10:00:00
stop1,stop3,09:05:00,11:00:00
stop2,stop3,10:05:00,11:00:00
```

In other terms, we use a more extensive description than the original structure since we prefer to list all possible direct pairs *i.e.* without considering connections. Thereby, to find a path between two stations we just have to browse the built file keeping only matching lines with a *filter*. Nevertheless, if the wanted stations are not directly linked together, it implies that a connection must be done somewhere. A *join* is thus required between two or more subsets. That is why we use the notion of routes in the specification to reduce significantly the size of these joinable subsets: only stops having two or more routes passing by are kept *i.e.* we only consider stops corresponding to connection stations.

Finally, we try to minimize the number of connections during trips. When we sort potential matching paths we first look at the number of connections before the time of travel. This method

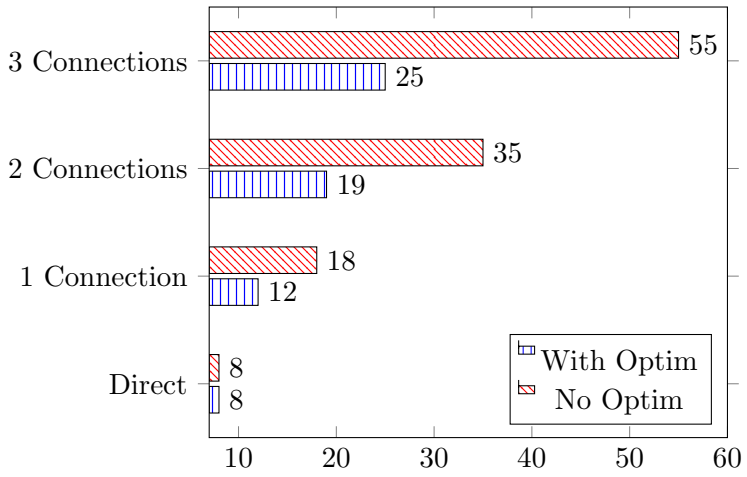


Figure 8.2: Comparison between the two methods using the *Ter* dataset. We present the average time (in seconds) to calculate journeys based on the number of connections.

implies that we do not guarantee the fastest possible travel; however, the path with the lower number of connections is often the fastest since having more connections always implies more waiting time. In practice, we try to minimize the number of joins during the querying step; therefore, adopting this policy allows also us to reduce the processing time of queries because joins are longer to compute than simple filters.

Therefore, to find a path between two stops, we first filter the dataset to find a direct path taking into account the times requested. If there is no result, we list the reachable connection stations (the locus where at least two GTFS routes intersect) from the starting point and then we retry the first step from each connection. This method can be done recursively until a path is found. If there are several possibilities we sort them by time.

To illustrate our point, let us consider the system drawn in Figure 8.1. It represents four lines which correspond to GTFS routes: L1 (A,B,C,D) in blue, L2 (E,C,F,G,H,I) in red, L3 (J,G,K,L,O) in cyan and L4 (K,M,N,O,P) in orange. In this system, there are 16 distinct stops and only 4 connections, thereby considering only them for the joins would reduce significantly subsets size. We can assume here that a train passes every five minutes in both directions at each stop.

We present here potential requested travels using the system introduced in Figure 8.1. First, to reach D from A, since it is direct with L1, paths are easy to compute. Then, from A to E, because these stops are not directly reachable, it is mandatory to look at their common reachable connections: here C is the only one. Finally, to access P starting at J, we have to list the connection stops J_c={G,K,O} and P_c={K,O}; then since $J_c \cap P_c$={K,O}, we have to choose the connection stop minimizing the trip duration: the trip passing through O is kept since it is 5 minutes faster (one stop less).

Datasets	Connection stations	Total of stations
France Transiliens	70	555
France Intercités	109	357
New York City MTA Queens	180	1 567
New York City Subway	117	377
San Francisco MTA	533	2 618
SNCB (Belgium)	609	631

Table 8.1: Connection ratio of some datasets.

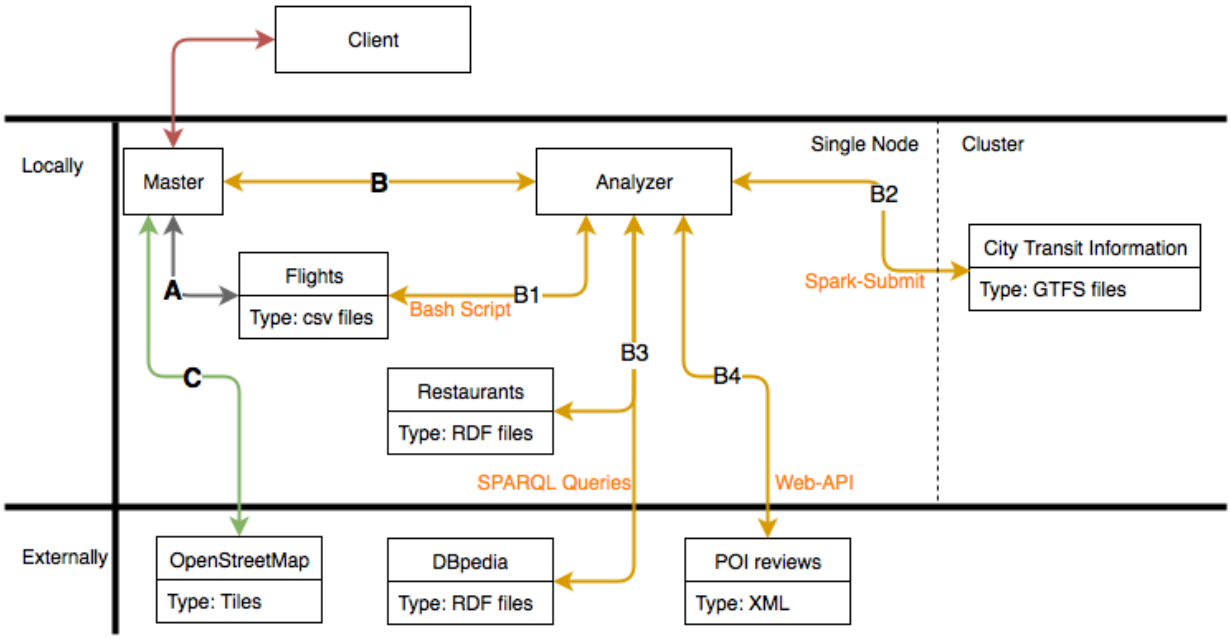


Figure 8.3: Overall Architecture.

Practically, we list in Table 8.1 some figures obtained using *real* data extracted from openly available GTFS datasets. As expected, in most datasets, the number of connection stations is small compared to the total number of stations *e.g.* there are only 180 connection stations in the NYC Queens dataset for a total of 1567. Then, joining only subsets having connection stations drastically reduces the size of considered objects and thus improves the execution speed. Indeed, as presented in Figure 8.2, in the *Ter* dataset ($\sim 6.10^5$ direct pairs), to find a journey with one connection, the optimized method allows a gain of 33%; the gain is even better if more connections are required: for intance, more than 50% with 3 connections.

8.3 Overall System Architecture

The global architecture of our system is shown in Figure 8.3. It consists of a lightweight client-side part in which users indicate a city of origin and a city of destination and which also displays results, and a backend part with an entry point called master. As shown in Figure 8.3, the master executes three different processes A,B and C. Process A corresponds to a usual flight finder: it returns trips sorted by simple criteria such as the number of connections and the transit time (by default). Process C queries the Open Street Map [49] tiles servers to fetch cartographic data for drawing resulting routes on a map. Processes A and C basically correspond to what can be found in common flight finding applications. The novelty of our idea and our system resides in process B, which is in charge of computing recommendations by reasoning on enriched data. This process performs the semantic searches, verifications, and filters that finally yields suggested smart trip options.

In the backend we distinguish datasets according to their sizes. For performance reasons, when datasets fit in main-memory of a single machine, we make sure that computations from the performances of in-memory engines. This is the case for the flight database and the restaurant databases that fit on a single node whereas city transit data is distributed across a cluster of nodes. Indeed, the sizes of these databases are not of the same order of magnitude. For example, for the Los Angeles (California) city area and its main airport, flight and restaurant data represent 2Mb,

whereas the size of public transportation data for the same area is closed to 20Gb (due to at least seventy million direct paths between all regional stations).

Our system store city transit information using the General Transit Feed Specification (GTFS) [45] datasets, which consist of several CSV files providing routes, schedules, stations and stop times for instance.

For the purpose of scalability with the increasing amount of GTFS data made available by transit agencies, we designed our GTFS store on top of Apache Spark[2]. We store GTFS data in terms of resilient distributed datasets (RDDs) [87] over which we issue queries with Spark's dataframe API.

To obtain smart trip alternatives for a transit between two airports A_1 and A_2, process B (see Figure 8.3) performs reasoning on aggregated data coming from various sources thanks to the four sub-processes B1, B2, B3, and B4 (shown on Figure 8.3).

First of all, the analyzer queries the flight finder (B1) to have all possible paths (without cycles) taking off from A_1 and landing at A_2. Then, it applies filters to this set of paths according to two default (customizable) usecases: (1) it keeps paths having at least a three-to-five hour connection; (2) it only considers paths having at least a connection longer than eight hours. Moreover, it always tries to avoid connections requiring to spend one night in a hotel somewhere on Earth and rather promotes night in aircrafts, by default.

Knowing the possible connections, the analyzer asks the GTFS store (B2) to find among the city transit datasets all the accessible areas from each intermediate airport. This concept of accessibility depends on the usecase *i.e.* the GTFS store only considers areas from which one can go and return in less than M minutes, where M is equal to the minimum between one quarter of the connection time and 2 hours. For instance, if the connection time equals 3 hours we do not select areas located more than 45 minutes round trip; and if this connection lasts more than 8 hours, public transport transit times are limited correspondly (*e.g.* to 2 hours).

A set of accessible stations (using public transport) is hence available for each possible connection. To enrich user experience (B3), the analyzer seeks points of interest (POIs) in these areas. It uses DBpedia [12] as semantic data provider since DBpedia extracts factual information from Wikipedia and converts it into RDF. Practically, the analyzer sets up SPARQL queries [46]. Figure 8.4 presents a typical SPARQL query that might be generated.

We have implemented various strategies to limit the number of results. First, the SPARQL query selects elements of type `dbpedia:Place` and returns for each place its name, a short abstract, and a picture if possible using the `OPTIONAL` SPARQL feature. In the same time, a language filter is applied on names and abstracts with `FILTER(lang...)`. Finally, it keeps only the five closest POIs (`LIMIT` and `ORDER BY`) around the station (`X` and `Y` as GPS coordinates) within a radius `RAD`. Since several treated stations might be close, we define the different `RAD` not to have overlapping areas; we thus guarantee that each POI belongs to only one accessible station. Meantime, the analyzer queries our local restaurant RDF dataset to suggest places to eat while users do some sightseeing.

Then, in (B4) the analyzer fetches ranks and reviews of other users concerning all accessible POIs. Such an operation aims at adding human opinion within the processing chain: having ranks allows an *a priori* POI classification. To refine the best trip option, the analyzer considers several parameters: the average rank of an area, the effective time that is spent in this area, and even the length of the various abstracts. All these considerations are used to obtain an overall score for each area; the analyzer can thereby choose among the best retained ones using a (customizable) score function.

Finally, the master joins obtained GPS coordinates with OSM map tiles and then sends back to the client the three best results: first the fastest trip minimizing the number of connections and the overall trip duration, second an interesting trip alternative taking advantage of a connection

[2]http://spark.apache.org/

```
SELECT ?name ?abstract ?picture
WHERE {
  ?c  rdf:type           dbpedia:Place .
  ?c  dbpedia:abstract   ?abstract .
  ?c  rdfs:label         ?name .
  ?c  geo:long           ?long .
  ?c  geo:lat            ?lat .
  OPTIONAL {
    ?c  dbpedia:thumbnail  ?picture .
  }
  BIND(((xsd:double(?lat)-X)^2)+
       ((xsd:double(?long)-Y)^2)
      ) AS ?dist
  FILTER(lang(?name) = ''LANG'')
  FILTER(lang(?abstract) = ''LANG'')
  FILTER(?dist < RAD)
}
ORDER BY ASC (?dist)
LIMIT 5
```

Figure 8.4: SPARQL query extracting from dbpedia the 5 closest POIs in language LANG located around a point whose GPS coordinates are (X,Y) within a radius of RAD.

lasting three to five hours, third another interesting alternative when more than eight hours can be spent somewhere.

8.4 Typical System Usage

The typical scenario consists in using our processing pipeline in order to obtain smart trip alternatives using various data sources: GTFS schedules, OSM tiles and DBpedia RDF data. One interest of such a tool relies on the fact that users can find alternatives using real data *e.g.* the scheduling grids are the ones used each day by official transit agencies and semantic data comes from DBpedia. For instance it is possible to review suggestions of trip alternatives to come to the conference.

8.4.1 Sample of Real Datasets

We preloaded GTFS datasets of various transport agencies around the world. We extracted most of them from the GTFS data exchange platform[3]. In terms of disk footprint, they represent more than 50Gb on the cluster. Table 8.2 summarizes information of some datasets, *e.g.* numbers of routes or direct pairs or also stops.

8.4.2 Step-By-Step Usage Example

System users can search for trip alternatives passing as argument two airports and the two allowed time lapses for connections. For instance, from Paris in France (`CDG` airport) to Honolulu in Hawaii USA (`HNL` airport) with the default usecases, the pipeline might propose at first (process A in Figure 8.3) to pass through San Francisco Ca since it is the fastest trip available, all connections via Los Angeles Ca or via other airports are longer:

[3]http://www.gtfs-data-exchange.com/

Datasets	Stops	Routes	Direct Pairs	Loaded Size (Gb)
Los Angeles Ca	14 992	148	74 570 202	19,8
San Francisco Ca	4 577	85	17 358 866	7
New York City	17 923	1 317	146 231 113	44
Paris, France	3 204	34	2 534 273	1

Table 8.2: Information of some datasets.

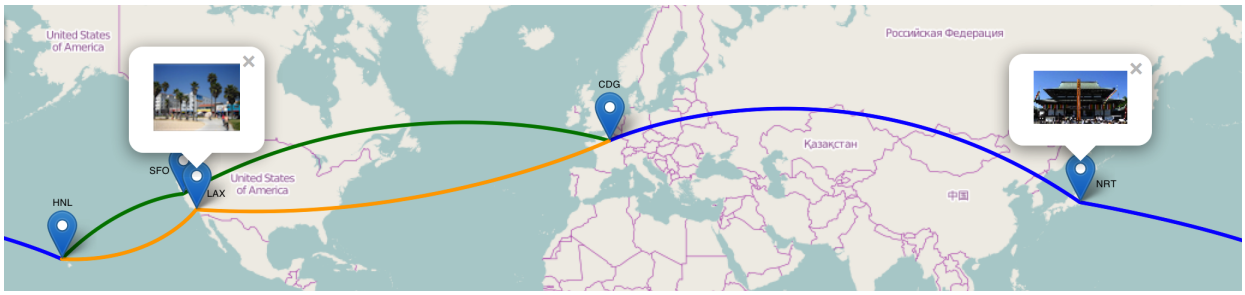


Figure 8.5: Application Screenshot (CDG→HNL).

```
== The 3 Fastest Trips ==
CDG (10:30--12:55) SFO (13:30--17:20) HNL
CDG (11:30--14:30) LAX (14:40--18:22) HNL
CDG (13:30--16:15) LAX (16:45--20:35) HNL
```

After acting as a conventional "fastest trip finder", the application computes smart alternative trips using the various tools previously presented (Section 8.3) and grouped in the process B in Figure 8.3. First of all, it searches trips having a three-to-five hour connection; considering the example introduced above (CDG→HNL), only 14 options are possible, and 4 airports can be used as connection: San Francisco Ca (SFO), Los Angeles Ca (LAX), Seattle Wa (SEA) and Tokyo Japan (NRT). These trips sorted in descending order of connection time are:

```
CDG (21:20--17:05) NRT (22:00--09:35) HNL
CDG (11:30--14:30) LAX (18:45--22:33) HNL
CDG (13:30--15:13) SEA (19:25--23:58) HNL
CDG (09:10--12:15) SFO (16:05--19:42) HNL
```

For each possible stopover, the GTFS store is used to identify all the viable accessible areas from the airports. Then the application looks for closest POIs around these areas. In the CDG→HNL case, it considers for instance the Naritasan Shinsho-ji Temple (20 minutes far from NRT) or also Venice Beach (40 minutes far from LAX). At the end, taking into account the effective time available in each possible sites and the ranking averages of various areas, our application suggests a stopover in Narita.

Then, a second alternative is presented to attendees. However, this option which should take advantage of a long connection time (more than eight hour long) does not appear to be valuable in the case CDG→HNL. Actually, each trip involves an overnight stay in a hotel, even if new intermediate destinations are available *e.g.* Atlanta Ga, Dallas Tx, Chicago Il, or Vancouver Canada. Rather than returning an empty result, the application uses the list made in the previous case (with a three-to-five hour connection) and returns the second most interesting trip *i.e.* passing through Los Angeles.

Finally, in addition to the fast Paris-Honolulu via San Francisco, two alternative journeys are proposed (Figure 8.5): one via Tokyo and another one via Los Angeles.

8.5 Conclusion

There exists many trip planning systems such as Google Maps [1] and Rome2Rio [2] for example. These systems allow one to easily obtain routes that satisfy simple criteria such as shortest path, shortest duration, cheapest price, and combinations of them. Compared to these systems, we bring an additional semantic layer that allows our system to suggest smart alternatives, *e.g.* alternatives that do not necessarily satisfy the initial criteria entered by the user, but that will be preferred in the end.

Using DBpedia [12] as a POI provider in a tourism context has been proposed by [26, 31]. We used DBpedia similarly in the more specific case of journey planning.

Closest to our approach are the works on automatic construction of travel itineraries [28, 33] and interactive itinerary planning [76]. These approaches typically look for feasible itineraries given a particular location and time budget. Specifically, the approach in [76] introduces an interactive planning process that starts with a user providing a time budget and a starting point of the itinerary (usually corresponding to the hotel where the user is staying). The system progressively suggests a touristic itinerary depending on successive user feedbacks until the user chooses a specific tour.

Compared to these approaches, we notably leverage the use of GTFS big data [45] for checking feasibility of the itinerary by public transportation. Furthermore, our generic architecture might benefit from the developments in [31, 76] for generating even more alternatives. Indeed our processes B3 and B4 shown on Figure 8.3 might also include such additional systems to improve the set of relevant alternatives.

Last but not least, an advantage of our system compared with [33, 76] is to provide alternatives at booking time. The user becomes active in the layover decision process deciding how and where to spend its time budget.

General Conclusion & Perspectives

We now summarize the advances we realized. Firstly, we conclude on our contributions. Secondly, we present a list of perspectives and future works that our thesis has led to. Thirdly, we recap the various projects that we made available online. Fourthly, we present the summary of our publications.

Contributions

We focus in this thesis on the evaluation of SPARQL queries in a distributed context *i.e.* when an RDF dataset is split over several nodes.

As shown in Chapter 3, there exists a large number of subproblems and an even larger variety of systems addressing them. Therefore, we reduce the scope of our study to clusters of nodes which already offer a distributed file system (*e.g.* the HDFS). We made this choice for two reasons: first, it provides a form of resiliency and second it allows to use powerful tools already developed especially for these file systems (*e.g.* Apache Hive).

In a second time, we selected a set of distributed SPARQL evaluators according to criteria such as recentness, availability of sources, performances... We then benchmarked them on our own cluster (see Chapter 4). This preliminary work aimed at offering us a large common basis of evaluation for the future evaluator we would build. However, it also appeared that the behaviors of systems were very different: for instance some were slow all the time whereas others were fast on most of the queries and failed to evaluate some others. Thereby, one of the basic need for our future system should be the regularity of evaluation.

In parallel, new tools were developed to realize cluster computation. Among them, Spark [87] offers a very efficient set of low-level primitives to compute operations on files with a MapReduce-like strategy. Moreover, it is able to take the HDFS as a file system. We thus constructed a translator of a fragment of SPARQL which outputs optimized – *e.g.* according to statistics on initial RDF data – Spark-compliant Scala-code.

We present in Chapter 5 this translator and the selected RDF storage strategy under the name of SPARQLGX. Moreover, we validate this new approach with an experimental validation.

When we selected the benchmarks that would be used during our experiments, we noticed that existing benchmarks only consider temporal metrics (and sometimes disk footprint) for measuring runtime performance. In addition, real-world applications seem not only to require temporal efficiency but also resiliency and/or resource consumption parsimony. That is why we developed in Chapter 6 a new reading grid based on five criteria namely: velocity, immediacy, dynamicity, parsimony and resiliency. We showed that depending on the interest in criteria the tested systems were ranked differently. For instance, we noted that SPARQLGX would be a reasonnable choice when all criteria but parsimony matter.

We next developed two additional SPARQL evaluators dedicated to the immediacy criteria in Chapter 7: RDFHive and SDE. They both are good choices if only a few SPARQL queries are executed once. Additionally, they both are resilient and can deal with dynamic data.

Finally, in Chapter 8, we presented a practical case of trip planning where heterogeneous semantic datasets from various sources (*i.e.* public transports, flights, restaurant lists) need to be queried, and the sub-results aggregated/merged to obtain a consistent trip proposal. This last Chapter provides an opportunity to see how SPARQL evaluators can be integrated into a larger project where querying distributed RDF datasets is not a goal but a tool.

Perspectives

During this thesis, we investigated some various research axes in the field of distributed SPARQL evaluation. Actually, we notably:

1. proposed a new reading grid to rank SPARQL evaluators
2. developed several evaluators
3. briefly presented a more general application using several heterogeneous semantic tools

We now present some perspectives for further developments along the above axes.

1. SPARQL Benchmarking

The work presented in Chapters 4 & 6 offers an experimental snapshot with state-of-the-art distributed SPARQL evaluators. In a sense, we updated – while adding a new reading grid – a work carried by Cudré *et al.* in [32]. Staying up to date will therefore require a rolling work in order to add new evaluators in the test suite.

Moreover, this experimental study can also be improved by extending the reading grid. Actually, we consider a fixed number of nodes in our cluster (in particular: 10) because of the price of hundreds/thousands of nodes and hard disks. Nevertheless, studying the *scalability* of the distributed evaluators would constitute an interesting step for providing a new dimension in the design a reading grid. Indeed, the 5 already considered criteria are all "orthogonal" to the scalability criteria since at a defined scale a "Chapter 6"-like analysis can be conducted. Additionally, the scalability would also point new possible limitations: for instance, it is hard to guess the behaviors of master/slaves architectures when there are hundreds of slaves: perhaps the cluster resources would have to be thousands times larger than with a 10-nodes cluster.

2. SPARQL Evaluators

The study on SPARQLGX presented in Chapter 5 can be continued and improved. A subsequent evolution can be found in the supported SPARQL fragment; actually, we mainly focused on optimizing BGPs and we then extended the translation to other SPARQL keywords (*e.g.* solution modifiers or `OPTIONAL` under specific conditions). Since SPARQLGX already represents an attractive tool, it would be interesting to tackle optimizations for more expressive query fragments.

More generally, criteria-specific distributed SPARQL evaluators can be implemented like the two ones already introduced in Chapter 7. For instance, having a very parsimonious resilient systems would perhaps help to evaluate queries in a highly competitive environment where lots of other applications/tools also need CPU and/or RAM and where the risk of a failure increases.

In [8], Aluç *et al.* consider that the next significant performance gain in the field of single-node SPARQL evaluators will be reached with evaluators able to adapt on-the-fly the storage structure used according to the query patterns. Such a method implies to pre-process datasets in order to store them using several approaches (see *e.g.* Chapter 3) at the same to allow the query evaluator to swich from one representation to an other. In a distributed context, where there exits multiple RDF approaches, a similar paradigm could be implemented starting from the discussion about variable graphs conducted in Chapter 6. Indeed, we already showed that some storage methods

should be recommended with specific variable graphs; we can thereby imagine storage-adaptative distributed SPARQL evaluators.

3. Integrating SPARQL evaluators in ETL systems

During our experiments, we faced the lack of test suite dedicated to SPARQL update primitives and had to design our own protocol. As a consequence, having a standardized benchmark dealing with these functionalities would allow for more reproducibility to the comparative tests realized.

In the development of our study, we focused on the SPARQL `SELECT` queries *i.e.* which extract a list of columns from a set of RDF triples. Using `CONSTRUCT` queries, it is possible to extract triples from an initial set of triples. Considering this statement, SPARQL treatment pipelines can be designed to refine – by enriching, cleaning, merging... – several distributed RDF sources into one unique set of triples which can be then query intensively after. In particular, ETL (extract, transform and load) chains of SPARQL queries can be developed. Technically, difficulties can be found in the management of updates coming from each distinct source since a new computation "from scratch" can be very expensive.

Finally, the case of SPARQL pipelines can also be extended to heterogeneous pipelines. For example, we presented such a pipeline in Chapter 8 as a proof a concept. More generally, a layer of formalism could be designed – encompassing the presented example of trip planning – in order to guarantee that the process will finish correctly or to improve automatically performances by *e.g.* changing the order of elementary operations. For instance, the formalism could imply a slight typing system on the resource nature in order to allow pipes and merges, more particularly, the various forms of SPARQL queries can be used as a starting point *e.g.* `SELECT` takes triples and returns columns, `ASK` takes triples and returns booleans.

External Links

- The detailed results of our experiments are displayed on `http:/tyrex.inria.fr/sparql-comparative/`.
- SPARQLGX and SDE are available online from `https://github.com/tyrex-team/sparqlgx`. We provide tutorials and instructions to install and configure them.
- RDFHive is also on the team's *github* at `https://github.com/tyrex-team/rdfhive`.
- The GTFS-store can be found at `https://github.com/tyrex-team/gtfs-store`.

Related Publications

- **SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark.**
  Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda.
  The 15th International Semantic Web Conference, Oct 2016, Kobe, Japan.
- **SPARQLGX in action: Efficient Distributed Evaluation of SPARQL with Apache Spark.**
  Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda.
  The 15th International Semantic Web Conference, Oct 2016, Kobe, Japan.
- **Smart Trip Alternatives for the Curious.**
  Damien Graux, Pierre Genevès, Nabil Layaïda.
  The 15th International Semantic Web Conference, Oct 2016, Kobe, Japan.

- **A Multi-Criteria Experimental Ranking of Distributed SPARQL Evaluators**
  Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda.
  ¡hal-01381781¿, 2016.

Bibliography

[1] Google-Maps. https://www.google.fr/maps.

[2] Rome2Rio. http://www.rome2rio.com/.

[3] RDF 1.1 N-Triples: A line-based syntax for an RDF graph, 2014. http://www.w3.org/TR/n-triples/.

[4] Abadi, Marcus, Madden, and Hollenbach. Scalable semantic web data management using vertical partitioning. *VLDB*, 2007.

[5] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *VLDB*, 18(2):385–406, 2009.

[6] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ics-forth rdfsuite: Managing voluminous rdf description bases. In *Proceedings of the Second International Conference on Semantic Web-Volume 40*, pages 1–13. CEUR-WS. org, 2001.

[7] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212. Springer, 2014.

[8] G. Aluç, M. T. Özsu, and K. Daudjee. Workload matters: Why rdf databases need a new design. *Proceedings of the VLDB Endowment*, 7(10):837–840, 2014.

[9] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev, and I. Toma. The linked data benchmark council: a graph and RDF industry benchmarking effort. *ACM SIGMOD Record*, 43(1):27–31, 2014.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[11] M. Atre, J. Srinivasan, and J. Hendler. Bitmat: A main-memory bit matrix of rdf triples for conjunctive triple pattern queries. In *Proceedings of the 2007 International Conference on Posters and Demonstrations-Volume 401*, pages 1–2. CEUR-WS. org, 2008.

[12] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. *DBpedia: A nucleus for a web of open data*. Springer, 2007.

[13] A. Barstow. Survey of rdf/triple data stores. *World Wide Web Consortium. Retrieved April*, 10:2003, 2001.

[14] D. Beckett. The design and implementation of the redland rdf application framework. *Computer Networks*, 39(5):577–588, 2002.

[15] D. Beckett. Scalability and storage: Survey of free software/open source rdf storage systems. *Latest version is available at http://www. w3. org/2001/sw/Europe/reports/rdf_ scalable_storage_report*, 2002.

[16] D. Beckett and J. Grant. Mapping semantic web data with RDBMSes. *W3C Semantic Web Advanced Development for Europe (SWAD-Europe)*, 2003.

[17] D. Beckett and B. McBride. Rdf/xml syntax specification (revised). *W3C recommendation*, 10, 2004.

[18] T. Berners-Lee. Artificial intelligence and the semantic web: Aaai2006 keynote. *W3C Web site*, 2006.

[19] T. Berners-Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[20] C. Bizer and A. Schultz. The berlin SPARQL benchmark. *IJSWIS*, 2009.

[21] D. Blum and S. Cohen. Grr: generating random rdf. In *Extended Semantic Web Conference*, pages 16–30. Springer, 2011.

[22] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 2007.

[23] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16:16, 1998.

[24] D. Brickley and R. V. Guha. {RDF vocabulary description language 1.0: RDF schema}. 2004.

[25] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, pages 54–68. Springer, 2002.

[26] A. E. Cano, G. Burel, A.-S. Dadzie, and F. Ciravegna. Topica: A tool for visualising emerging semantics of pois based on social awareness streams. In *10th Int. Semantic Web Conf (ISWC2011)(Demo Track)*, 2011.

[27] G. Carothers and E. Prud'hommeaux. RDF 1.1 turtle. Feb. 2014.

[28] G. Chen, S. Wu, J. Zhou, and A. K. Tung. Automatic itinerary planning for traveling services. *Knowledge and Data Engineering, IEEE Transactions on*, 26(3):514–527, 2014.

[29] U. Consortium et al. *The Unicode Standard, Version 2.0*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[30] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[31] S. Cresci, A. D'Errico, D. Gazzé, A. Lo Duca, A. Marchetti, and M. Tesconi. Towards a DBpedia of tourism: the case of tourpedia. In *Proceedings of the 2014 International Conference on Semantic Web-Poster and Demo Track, ISWC2014*, pages 129–132, 2014.

[32] P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. F. Sequeda, and M. Wylot. NoSQL databases for RDF: An empirical evaluation. *ISWC*, pages 310–325, 2013.

[33] M. De Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu. Automatic construction of travel itineraries using social breadcrumbs. In *Proceedings of the 21st ACM conference on Hypertext and hypermedia*, pages 35–44. ACM, 2010.

[34] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[35] M. Dean, G. Schreiber, S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Owl web ontology language reference. *W3C Recommendation February*, 10, 2004.

[36] G. Demartini, I. Enchev, M. Wylot, J. Gapany, and P. Cudré-Mauroux. Bowlognabench – Benchmarking RDF Analytics. In *International Symposium on Data-Driven Process Discovery and Analysis*, pages 82–102. Springer, 2011.

[37] C. Doulkeridis and K. Nørvåg. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, 2014.

[38] O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[39] D. C. Faye, O. Curé, and G. Blin. A survey of RDF storage approaches. *Arima Journal*, 15:11–35, 2012.

[40] L. Galarraga, K. Hose, and R. Schenkel. Partout: A distributed engine for efficient rdf processing. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, pages 267–268. International World Wide Web Conferences Steering Committee, 2014.

[41] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. In *1st International Workshop on Usage Analysis and the Web of Data at the 20th International World Wide Web Conference*, 2011.

[42] A. Gerber, A. Van der Merwe, and A. Barnard. A functional semantic web architecture. In *European Semantic Web Conference*, pages 273–287. Springer, 2008.

[43] F. Goasdoué, Z. Kaoudi, I. Manolescu, J.-A. Quiané-Ruiz, and S. Zampetakis. Cliquesquare: Flat plans for massively parallel RDF queries. In *ICDE*, pages 771–782. IEEE, 2015.

[44] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye. Building linkedin's real-time activity data pipeline. *IEEE Data Eng. Bull.*, 35(2):33–45, 2012.

[45] Google. GTFS definition, September 2006. https://developers.google.com/transit/gtfs/.

[46] W. S. W. Group et al. SPARQL 1.1 overview, 2013. http://www.w3.org/TR/sparql11-overview/.

[47] R. Guha. rdfdb: An rdf database, 2000.

[48] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics*, 2005.

[49] M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *Pervasive Computing, IEEE*, 7(4):12–18, 2008.

[50] S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage. 2003.

[51] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. *SSWS*, 2009.

[52] A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *Third Latin American Web Congress (LA-WEB'2005)*, pages 10–pp. IEEE, 2005.

[53] P. Hayes and B. McBride. RDF semantics. *W3C recommendation*, 10, 2004. www.w3.org/TR/rdf-concepts/.

[54] I. Horrocks, B. Parsia, P. Patel-Schneider, and J. Hendler. Semantic web architecture: Stack or two towers? In *International Workshop on Principles and Practice of Semantic Web Reasoning*, pages 37–41. Springer, 2005.

[55] K. Hose and R. Schenkel. Warp: Workload-aware replication and partitioning for rdf. In *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*, pages 1–6. IEEE, 2013.

[56] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

[57] Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *The VLDB Journal*, 24(1):67–91, 2015.

[58] M. Kifer and H. Boley. Rif overview. *W3C*, 2010.

[59] D. Kolas, I. Emmons, and M. Dean. Efficient linked-list rdf indexing in parliament. *SSWS*, 9:17–32, 2009.

[60] G. Ladwig and A. Harth. CumulusRDF: linked data management on nested key-value stores. *SSWS 2011*, page 30, 2011.

[61] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[62] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: Mapreduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, 2012.

[63] R. Lee. Scalability report on triple store applications. *Massachusetts institute of technology*, 2004.

[64] J. Leibiusky, G. Eisbruch, and D. Simonassi. *Getting started with storm.* " O'Reilly Media, Inc.", 2012.

[65] A. Magkanaraki, G. Karvounarakis, T. T. Anh, V. Christophides, and D. Plexousakis. Ontology storage and querying. *Ics-forth Technical Report*, 308, 2002.

[66] L. Masinter, T. Berners-Lee, and R. T. Fielding. Uniform resource identifier (uri): Generic syntax. 2005.

[67] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. DBpedia SPARQL Benchmark – Performance assessment with real queries on real data. *ISWC*, pages 454–469, 2011.

[68] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[69] M. Odersky. The scala language specification v 2.9, 2014.

[70] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110. ACM, 2008.

[71] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer, 2006.

[72] E. PrudHommeaux, A. Seaborne, et al. SPARQL query language for RDF. *W3C recommendation*, 15, 2008. www.w3.org/TR/rdf-sparql-query/.

[73] R. Punnoose, A. Crainiceanu, and D. Rapp. RYA: a scalable RDF triple store for the clouds. In *International Workshop on Cloud Intelligence*, page 4. ACM, 2012.

[74] S. Qiao and Z. M. Özsoyoğlu. Rbench: Application-specific RDF benchmarking. In *SIGMOD*, pages 1825–1838. ACM, 2015.

[75] A. Reggiori and D. van Gulik. Rdfstoreperl api for rdf storage. *Online only*, 2004.

[76] S. B. Roy, G. Das, S. Amer-Yahia, and C. Yu. Interactive itinerary planning. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 15–26. IEEE, 2011.

[77] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to pig latin. In *Proceedings of the International Workshop on Semantic Web Information Management*, page 4. ACM, 2011.

[78] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF querying with SPARQL on spark. *VLDB*, pages 804–815, 2016.

[79] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^2Bench: a SPARQL performance benchmark. *ICDE*, pages 222–233, 2009.

[80] M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*, pages 4–33. ACM, 2010.

[81] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[82] F. Stegmaier, U. Gröbner, M. Döller, H. Kosch, and G. Baese. Evaluation of current rdf database solutions. In *Proceedings of the 10th International Workshop on Semantic Multimedia Database Technologies (SeMuDaTe), 4th International Conference on Semantics And Digital Media Technologies (SAMT)*, pages 39–55. Citeseer, 2009.

[83] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[84] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146. ACM, 1982.

[85] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[86] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *Proceedings of the First International Conference on Semantic Web and Databases*, pages 120–139. CEUR-WS. org, 2003.

[87] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*, 2012.

[88] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. Scope: parallel databases meet mapreduce. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(5):611–636, 2012.